VisualSPARK 1.0 Installation & Usage Guide: Windows

Simulation Problem Analysis and Research Kernel

Lawrence Berkeley National Laboratory
Ayres Sowell Associates, Inc.

Contents

List of Figures

List of Tables

Foreword

This document is a *VisualSPARK* Windows 95/98/NT platform supplement to the SPARK User's Manual. It addresses platform specific issues, and provides installation instructions for these platforms. There are similar documents for *VisualSPARK* on UNIX platforms, and for the alternative Windows 95/98/NT platform interface called *WinSPARK*.

The SPARK User's Manual should be read before reading this document.

This work was supported by the Assistant Secretary for Energy Efficiency and Renewable Energy, Office of Building Technology, State and Community Programs, Office of Building Systems of the U.S. Department of Energy, under contract DE-AC03-76SF00098.

1 Introduction

SPARK is a general software program for solving simulation problems. It can be run on a variety of platforms. User interfaces for SPARK are considered to be separate software programs which, except for the built-in command-line interface, tend to be platform specific. The initial release of SPARK targets two platforms, namely UNIX and Intel based platforms running Microsoft Windows 95, 98 or NT. Each has one or more graphical user interfaces. This document deals with a graphical user interface called *VisualSPARK* for Windows platforms. The *VisualSPARK* interface is copyright by the Regents of the University of California.

The SPARK User's Manual provides overview, tutorial examples, and language reference that are, to the extent possible, platform independent. This document supplements the User's Manual, giving installation and usage information specific to *VisualSPARK* on Windows platforms. A similar document, the *VisualSPARK Installation and Usage Guide: UNIX,* is available for UNIX platforms. Additionally, there is an installation and usage guide document for *WinSPARK*, an alternative Windows interface for SPARK.

2 Availability and Licensing

VisualSPARK is now available to authorized persons. To obtain permission to install the software you must execute a Licensing Agreement. You can execute the Licensing Agreement by visiting the Simulation Research Group Web site at http://SimulationResearch.lbl.gov as discussed under Installation below.

Binary files that are part of the SPARK distribution are hardware and system software dependent. Initially, the *VisualSPARK* for Windows distribution is available for use with the Cygnus environment, *CygWin*. The *CygWin* environment is available for download from various Web sites.

3 Downloading

VisualSPARK is available only through downloading over the Internet. Detailed download and installation instructions are available at:

http://SimulationResearch.lbl.gov/VisualSPARK/download.htm

3.1 Naming convention

The naming convention for the download file is:
VisualSPARK<nnn>_<OS>_<CPU>.exe

where <nnn> is the *VisualSPARK* distribution version, <OS> is your operating system, and <CPU> is your CPU designation.

For example, the initial distribution for Windows on PCs is *VisualSPARK100_Win_x86.exe*.

3.2 Procedure

As explained in the detailed instructions provided at the above site, the download normally consists of two steps.

3.2.1 Download the CygWin GNU environment

This online document also contains links to the Cygnus site for convenient download of the required *CygWin* environment. The download file is called *full.exe* and is approximately 14 megabytes in size. If you already have the *CygWin* environment installed, you can skip this step. Store the downloaded files in a temporary directory on your machine.

3.2.2 Download VisualSPARK

The download file is called *VisualSPARKnnn_Win_x86.exe*, where *nnn* is the current release version number. The size of the download file is approximately 10 megabytes. Store the downloaded files in a temporary directory on your machine, and proceed to Installation below.

4 Installation

4.1 Requirements

VisualSPARK requires 40 megabytes of disk space for Windows installation, including the user interface. The *CygWin* environment occupies approximately 54 megabytes of disk space. These sizes will vary depending upon the file system you use, i.e., FAT, FAT32, or NTFS.

Although *VisualSPARK* will run small problems with as little as 8 megabytes of RAM, solution speed and maximum problem size will be strongly affected by processor speed and the amount of physical memory available. On Windows platforms, 128 megabytes of RAM is recommended. If slow solution speed is accompanied by excessive disk activity, a need for more memory may be indicated.

4.2 Installation Procedure

4.2.1 Installing CygWin

You must install the *CygWin* environment before installing *VisualSPARK*. The automated installation process for this software is similar to other Windows software. Simple go to the temporary directory where you saved the downloaded *full.exe* file and execute it, e.g., by clicking on it in Windows Explorer or by using the Windows “Start | Run” menu choice. This initiates the installation process. You will be given the opportunity to select the installation directory.

4.2.2 Installing VisualSPARK

The *VisualSPARK* download file, *VisualSPARKnnn_Win_x86.exe,* is a password protected, compressed “self installing” file. When this file is executed, you will be prompted for the password obtained when you executed the Licensing Agreement.[1]

After you have the needed password, go to the directory where you stored *VisualSPARKnnn_Win_x86.exe* and execute it, e.g., by clicking on it in Windows Explorer or by using the Windows “Start | Run” menu choice. You will be given the opportunity to choose the disk drive on which to install *VisualSPARK* in the *\vspark* directory.[2]

The installation creates the necessary directories into which the *VisualSPARK* files will be uncompressed. It also places a *VisualSPARK* group on the Windows “Start | Programs” menu. This group contains:

- *VisualSPARK*, which launches the *VisualSPARK* graphical user interface.
- *SPARK Console*, which launches an MSDOS window and initializes it as needed for running SPARK at the command line, and

Also, an “uninstall *VisualSPARK*” entry is created in the Windows menu “Start | Settings | Control Panel | Add / remove programs”.

4.3 Setting the SPARK Environment

The *VisualSPARK* for Windows implementation of SPARK, like the UNIX version, uses environment variables to identify various directories needed to run SPARK. These are shown in Table 1, where *d* refers to the selected installation drive letter.

When working at the command line, these variables can be set to the Table 1 values by executing the *sparkenv.bat* command procedure:

```
d:\vspark> sparkenv <enter>
```

[1] See http://SimulationResearch.lbl.gov/VisualSPARK/download.htm.

[2] In this release, you can specify the installation drive, but not the path. *VisualSPARK* must be installed in the *\vspark* directory of the selected drive.

You should not change the default values of these variables unless you are changing the installed location of *VisualSPARK* on your computer.[3]

Table 1 SPARK Environment Variables

Environment variable	**Definition**	**Default value** (Set during installation)
SPARK_DIR	The SPARK directory path, UNIX style.	d:/vspark
SPARK_DIRW	The SPARK directory path, Windows style.	d:\vspark
PATH	Windows PATH environment variable.	d:\vspark\bin;d:\vspark\gccmingw\bin; %PATH%
MAKE_MODE	Sets mode for the *gmake* program.	UNIX

In addition to the above environment variables, *VisualSPARK* relies on two files containing other information about the installation and usage environment, as shown in Table 2. These files are created with the default content by the installation process. However, the *classpath.env* file needs to be changed whenever your current problem uses classes stored in different directories, or when you want to change the order in which the directories are searched for classes used in your problem. You can use any Windows text editor to change this file. The *projects.env* file is maintained entirely by *VisualSPARK*; you should not edit it. It is not used when you are executing SPARK at the command line.

Table 2 SPARK Enviornment Files

Environment File	**Definition**	**Default content** (Set during installation)
classpath.env	Contains the current SPARK class path, i.e., list of directories to be searched for SPARK classes.	SPARK_CLASSPATH=.,../class,d:/vspark/globalclass,d:/vspark/hvactk/class
projects.env	Used by the *VisualSPARK* graphical user interface to manage the current list of project directories.	VSPARK_PROJS=d:\vspark\doc\examples

5 Documentation

SPARK documentation is in the v*spark\doc* directory after installation is complete. The User's manual is provided in PDF format in *sparkuser10.pdf.* Additionally, this Installation & Usage Guide is provided as *vswin_iug.pdf* in the *vspark/doc directory* . These documents can be viewed from within *VisualSPARK* by using the Help menu. Externally, you can open the PDF files with Acrobat Reader, available from various Internet sites. If you prefer paper copies, you can print these documents with Acrobat Reader. For improved navigation of the PDF documents, the option that shows the TOC should be set in PDF viewer. For AcroRead, set the option "Bookmarks and Page" in the View menu to turn on the TOC. For Acrobat 4.0, set “Bookmarks” in the Window menu to turn on the TOC.

The *vspark\doc* directory also has several text files with useful information. These files often contain information that became available after other documentation was complete, as well as currently known bugs and idiosyncrasies.

[3] If you do want to move *VisualSPARK* to a different drive, consider uninstalling and reinstalling it. This will ensure that all environment variables are properly set.

6 The SPARK Directory Structure

SPARK is composed of many files kept in several directories as shown in Figure 1. The *vspark\bin*, *vspark\lib* and *vspark\visspark* directories contain the fixed executable code and binary libraries[4] used by SPARK and the graphical user interface. Necessary source code header files are in *vspark\inc.* The *vspark\doc* directory contains user reference documents for SPARK and *VisualSPARK*.

SPARK comes with two application class libraries, the global classes and the HVAC tool kit classes. The global classes are in *vspark\globalclass* and include the basic mathematical functions likely to be needed in many SPARK problems, regardless of application area. The HVAC tool kit classes in *vspark\hvactk* define a modest library of classes for modeling heating, ventilation, and air-conditioning systems. The *vspark\doc\examples* directory and its subdirectories contain examples of SPARK problems using the global classes. Additionally, there is a compressed file in the *vspark\hvactk* directory that contains sample problems for all classes in the HVAC library. A utility, *testhvac*, is provided to extract, build, and execute these sample problems.

In *VisualSPARK*, each new problem must be in its own directory, called a *project* directory, that contains the problem file (*problem.pr*) and related files such as input (*problem.inp*), output (*problem.out*), and log files. Problems that are related are often grouped under a common parent directory, called the *projects* directory (plural). The *projects* directory, which can have any name, should also have a *class* subdirectory to hold any new classes you may create for the various problems under *projects*. The *vspark\doc\examples* subdirectory is an example of a *projects* directory. As can be seen in Figure 1, there are five *project* subdirectories under *examples*: *2sum, 4sum, exp1, frst_ord,* and *room_fc*.

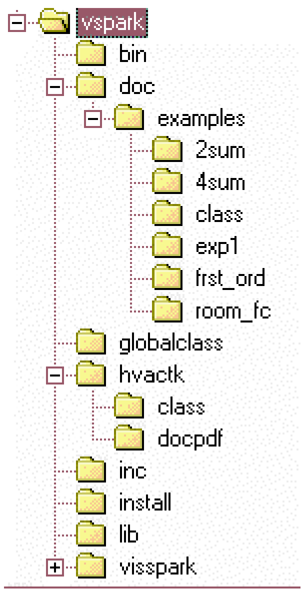


Figure 1 SPARK directory structure for VisualSPARK Windows installations.

When using the *VisualSPARK* interface the directory structure is extended by the addition of one or more *input set* subdirectories under each project directory. This is to allow you to have more than one set of inputs for each problem. *VisualSPARK* places input, output, and other files that are specific to a particular input set in the input set directory.

[4] Source is not provided for modules that are provided in binary form in the *bin* and *lib* subdirectories.

7 Command Line Execution of SPARK

7.1 Commands

With *VisualSPARK*, you can execute SPARK either with the *VisualSPARK* graphical user interface, or with commands issued in an *MSDOS* command window. In this section we focus on the *MSDOS* command method. See Section 8 for the *VisualSPARK* graphical user interface.

It is important to note that the following commands, e.g., *gmake* and *runspark*, do not have an argument indicating a particular problem to be run. This is because it is assumed that there is only one file in the project directory with the *.pr* extension. Consequently, if you want to have different versions of your problem you must place them in different project directories.

7.1.1 Preparations

As noted above (See Section 4.3), running a SPARK problem from the command line requires that the *vspark\bin* and certain other directories be in your PATH. Under Windows, you can accomplish this by opening an *MSDOS* window and executing the *sparkenv* command from your *vspark* directory:

```
    d:\> cd vspark <enter>
    d:\vspark> sparkenv <enter>
```

Alternatively, you can just select SPARK Console from the Windows "Start | Programs | VisualSPARK" menu choices. This launches an *MSDOS* window, changes to the *vspark* root directory, and automatically runs the *sparkenv* command file.

There are two additional tasks preliminary to running a SPARK problem at the command line. One is to be sure that the *classpath.env* file contains the correct path for the classes used in the problem. If you are using only the provided classes, the default values shown in Table 2 will be sufficient. Otherwise, use your text editor to modify *classpath.env* to include paths to your classes.

The other task is to provide a *makefile* in the project directory. One way to do this is to simply copy the master make file, *makefile.prj*, from the *vspark\lib* directory to the project directory:

```
d:\vspark> cd doc\examples\2sum <enter>
d:\vspark\doc\examples\2sum> copy d:\vspark\lib\makefile.prj makefile <enter>
```

Or, since *CygWin* provides an emulation of the UNIX symbolic link command, you could instead link to the master makefile:[5]

```
d:\vspark\doc\examples\2sum> ln -s d:\vspark\bin\makefile.prj makefile
<enter>
```

7.1.2 Build and Run

Once you have carried out these tasks, you are ready to run SPARK to solve your problem. This can be done with the *gmake*[6] command issued from the project directory. The command to run the problem in *prob_directory* is then:

```
d:\vspark\doc\examples\2sum> gmake run <enter>
```

This *gmake* command will use either the local *makefile*, or the link to the master *makefile.prj,* to orchestrate the various steps needed to build the executable "solver" program, and run that program to solve the problem (See Appendix, page 41, for details).

7.1.3 Run Control Information

When a SPARK problem runs it needs run-control information, such as simulation start time, time step, finish time, and locations of the needed input files and of the output file. In *VisualSPARK*, this information is taken from a file called *problem.run*. One of the things done by the above *gmake run* command is to

[5] In fact, most UNIX commands are provided with *CygWin*, e.g., *rm*, *mv*, and *cat*.

[6] Herein we shall assume that GNU make is called *gmake*, although some installations name it *make*. Special features needed in a SPARK build preclude use of most other *make* programs.

automatically create a *problem.run* file for the problem in the project directory. Should you wish to change the run-control information, you can edit *problem.run*.

7.1.4 Results

Results of the run may be found in the *project* directory. The produced files include:

- *problem.exe*: The executable solver file.
- *problem.prf*: Preference file used by the executable solver.
- *problem.eqs*: The equations file.
- *parser.log*: Log file for the parsing step.
- *setup.log*: Log file for the setup step.
- *run.log*: Log file for the execution step.
- Various intermediate files, usually of no concern to the user.

You should always check the log files to be sure no errors were encountered. The parser log file will show any problems in syntax as well as certain other errors in problem formulation. The setup log records errors in problem formulation not caught by parsing. The run log reveals errors during the execution phase, such as numerical problems. It also shows intermediate output of the solution process.

The equations file shows the calculation sequence followed by SPARK to solve the problem. It is explained in detail in the User's Manual.

7.1.5 The Runspark *Command*

To simplify the command line SPARK solution procedure described above, there is a command file called *runspark.bat* in directory *vspark\bin*. This command file is run by typing (in the current project directory):

```
d:\vspark\doc\examples\2sum> runspark <enter>
```

It will make the symbolic link to the SPARK master make file, supply the default *classpath.env* file, and run the problem in the current directory. To get the usage information about *runspark* type:

```
d:\vspark\doc\examples\2sum> runspark –help <enter>
```

To clean current project directory type:

```
d:\vspark\doc\examples\2sum> runspark  clean <enter>
```

This removes all files created during a previous run.

7.1.6 Rerunning a Problem Executable

As noted above, once you have run the problem by either of the above methods, the executable file named *problem.exe* will be in the project directory, along with a problem preference file, *problem.prf.* The *problem.prf* file contains the numerical solution settings for each component of the problem (See Appendix). If it is desired to run the numerical solution stage again, it can be initiated by executing the *problem* file with *problem.prf* and *problem.run* as arguments:

```
d:\vspark\doc\examples\2sum> 2sum 2sum.prf 2sum.run <enter>
```

Alternatively, you can again enter the *gmake run* command with little loss of efficiency, since *gmake* will not rebuild the problem if nothing has changed. On the other hand, if you do make changes to any of the classes or the problem file, you must use *gmake* to rebuild.

7.2 Examples

As can be seen in Figure 1, there are several project subdirectories under *vspark\doc\examples*. These contain problem specification files (*.pr*) and input files (*.inp*) from the examples in the User's Manual. A good one to start with is *4sum.pr,* discussed in Section 3 of the User's Manual. This problem simply adds four inputs, *x1, x2, x3,* and *x4,* with values taken from a provided input file *4sum.inp,* producing their sum, *x7*.

To run this example using the detailed steps indicated in the previous section, go to the *4sum* directory and set the class path and symbolic link to the SPARK make file:

```
d:\> cd d:\vspark\doc\examples\4sum <enter>
d:\vspark\doc\examples\4sum> ln -s d:\vspark\lib\makefile.prj makefile
<enter>
```

The next step is to build the solver for the problem and run the problem:

```
d:\vspark\doc\examples\4sum> gmake run <enter>
```

This results in creation of an executable program called *4sum.exe*. Several other files are created, including *4sum.prf* which is needed to execute *4sum,* and *4sum.run* which provides run-control information. The command also runs the executable. When the command completes, the results can be found in *4sum.out*, which can be examined with *Notepad* or another Windows or MSDOS editor. You should also examine the various log files noted in the previous Section.

We see that the inputs for x1 through x4, read from *4sum.inp*, were all 1, so their sum is 4. As before, these results are also placed in *4sum.out.*

7.3 Using SPARK Output

SPARK output files, i.e., *problem.out*, can be viewed with any text editor or viewer available on your computer. However, if the output is voluminous, such as for a dynamic simulation over a long time period, output is better viewed graphically, perhaps using a spreadsheet or other program that you may have installed.

8 Using the VisualSPARK Graphical User Interface

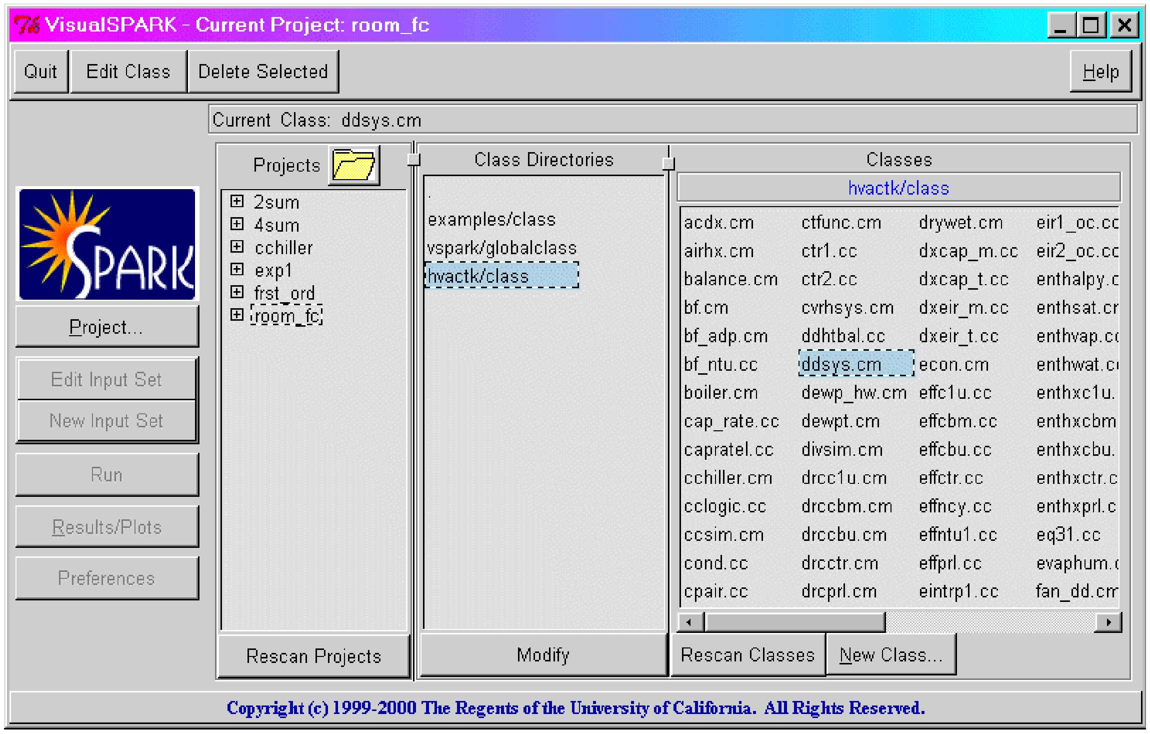
8.1 The Main VisualSPARK Window

The *VisualSPARK* graphical user interface for the Microsoft Windows environment is available as a more user-friendly environment than the command line. You can start *VisualSPARK* in either of two ways:

- You can go to the *vspark* root directory and type:
  `d:\vspark>` **`visspark <enter>`**
- You can select *VisualSPARK* from the Windows “Start | Programs” menu.

Either method will open the *VisualSPARK* main window on your Windows desktop as shown in Figure 2.

This screen has three principal panels, labeled Projects, Class Directories, and Classes, as well as command menu bars across the top and down the left side. The commands available in the menu bars will change depending upon which panel is active. A panel becomes active when you click your left mouse button while the cursor is in it. When the cursor is on a menu bar button, a brief description of what it does is presented in a pop-up window.



Current Class: ddsys.cm

Projects: 2sum, 4sum, cchiller, exp1, frst_ord, room_fc

Class Directories: . examples/class vspark/globalclass hvactk/class

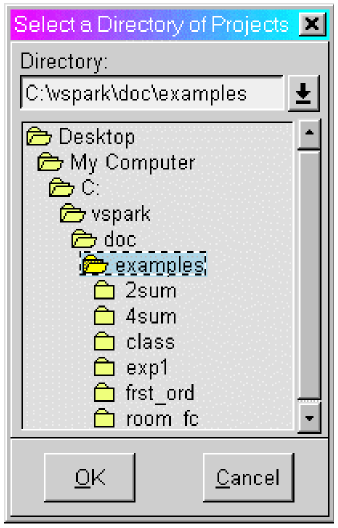
hvactk/class

acdx.cm	ctfunc.cm	drywet.cm	eir1_oc.cc
airhx.cm	ctr1.cc	dxcap_m.cc	eir2_oc.cc
balance.cm	ctr2.cc	dxcap_t.cc	enthalpy.c
bf.cm	cvrhsys.cm	dxeir_m.cc	enthsat.cr
bf_adp.cm	ddhtbal.cc	dxeir_t.cc	enthvap.cc
bf_ntu.cc	ddsys.cm	econ.cm	enthwat.c
boiler.cm	dewp_hw.cm	effc1u.cc	enthxc1u.
cap_rate.cc	dewpt.cm	effcbm.cc	enthxcbm
capratel.cc	divsim.cm	effcbu.cc	enthxcbu.
cchiller.cm	drcc1u.cm	effctr.cc	enthxctr.c
cclogic.cc	drccbm.cm	effncy.cc	enthxprl.c
ccsim.cm	drccbu.cm	effntu1.cc	eq31.cc
cond.cc	drcctr.cm	effprl.cc	evaphum.
cpair.cc	drcprl.cm	eintrp1.cc	fan_dd.cm

Copyright (c) 1999-2000 The Regents of the University of California. All Rights Reserved.

Figure 2 VisualSPARK main menu.

The Projects panel shows currently available projects upon which you can work. At the top of this panel there is an open folder icon, symbolizing a currently active Project Directory. Holding the cursor over the word "Project" in front of the folder icon causes the complete, current project path to be displayed in a pop-up window. Clicking on the folder opens a directory tree showing where this project folder is placed in your Windows file system, Figure 3.



Directory: C:\vspark\doc\examples

Figure 3 Projects directory tree.

If you wish, you can change to a different Projects directory by clicking on the desired folder in the tree.

Typically, a Projects folder will have several Project subdirectories with individual projects, i.e., SPARK problems. In turn, each project can have one or more subdirectories, e.g., *2sum_inp* below *2sum*. These subdirectories represent particular "input sets" for the project, so you can run the problem with different input data and run-control information. **In order to execute SPARK, one of these input set directories must be selected.**

The Class Directories panel shows class directories currently available for use in the project selected in the Projects panel. These are initially set to the classes defined in *classpath.env* file (See Section 4.3), but these paths may be rearranged or new paths added with the Modify button at the bottom of the panel. Once changed, the new class path list gets saved with the project information.

The Classes panel shows the classes in the directory currently selected in the Class Directory panel. Double clicking, or selecting and pressing the Edit Class button, for one of these classes opens the class file for editing.

Note the Rescan buttons below the Projects and Classes panels. While *VisualSPARK* can often automatically update the panel displays for changes made, it may not be aware of certain changes, e.g., adding a new project. The Rescan button forces panel update to deal with these situations.

The command menu across the top of the *VisualSPARK* main screen offers functions related to controlling *VisualSPARK*, such as Quit and Help, or editing classes and projects. Quit exits the *VisualSPARK* interface, while Help offers a menu with selections for various on-line SPARK documents as seen in Figure 4.

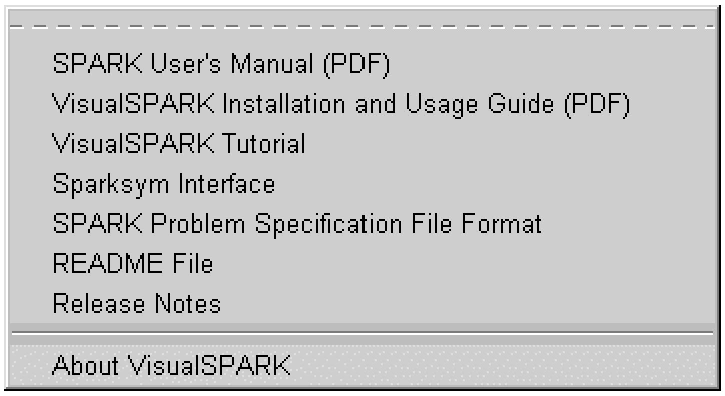

SPARK User's Manual (PDF)
VisualSPARK Installation and Usage Guide (PDF)
VisualSPARK Tutorial
Sparksym Interface
SPARK Problem Specification File Format
README File
Release Notes
About VisualSPARK

Figure 4 Help Menu

The Text Editor button applies to either Projects or Classes.[7] It executes a text editor and loads the selected file for editing. The Delete Selected button will delete the selected file, whether it is a project, an input set, or a class. The Graphical Editor[8] button launches a tool that allows construction of SPARK macro classes and problems in a graphical environment.

The menu at the left of the *VisualSPARK* window offers functions related to execution or other operations with SPARK projects. The buttons available at any time are determined by what is selected in the three panels. For example, if a project is selected in the Projects panel, only the Projects and New Input Set buttons are available. Clicking on the Projects button opens the Projects menu, Figure 5 that allows creating or copying a project, as well as other options. The New Input Set button allows you to create input data for the selected project. On the other hand, if an input set is selected in the Projects panel, the buttons for editing the input set, running the project and plotting results or changing preferences become available. These menu commands are explained in more detail in subsequent sections.

[7] The label on this button changes depending upon cursor location. When in the Projects panel the label is Edit Project, and it changes to Edit Class when you move to the Class panel. When the cursor is not in either of these panels, the button is inactive and labeled Text Editor.

[8] The Graphical Editor is not available in the initial release of *VisualSPARK*.

8.2 The Projects Menu

8.2.1 Creating and Copying Projects

Clicking on the Projects button opens the Projects menu as seen in Figure 5. From this menu you can create a new project either by starting fresh, or by copying an existing project.

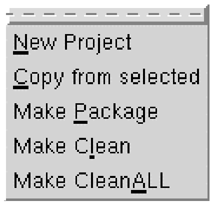
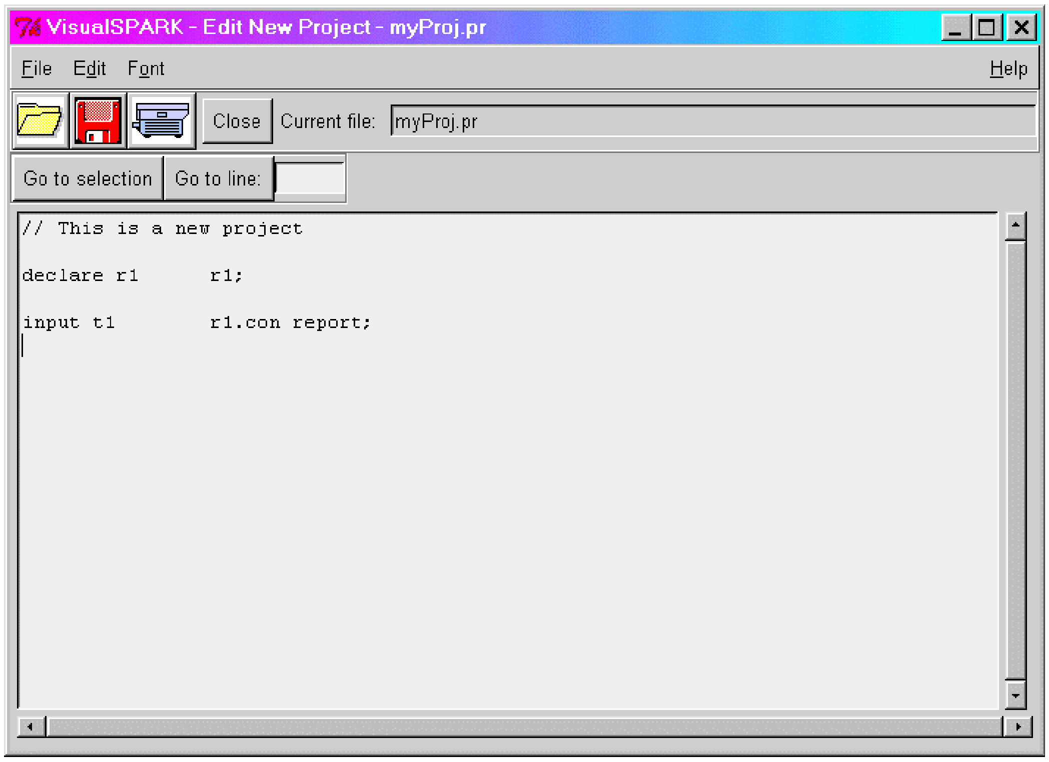


Figure 5 Projects menu.

Note that any new project created will be placed under the Projects directory currently appearing in the Projects panel. Therefore you should be sure you are in the correct Projects directory before clicking the Projects button (See Section 8.1).

If you select New Project, you will be asked for a new project name in a dialog, then a text editor will be opened with an empty file, Figure 6. After typing the SPARK problem definition into this file, saving will create a new subdirectory for it in the Projects directory and the new project will then appear in the Projects panel.



```
// This is a new project

declare r1      r1;

input t1        r1.con report;
```

Figure 6 New Project text editor.

If you want to create a new project based on an existing problem file, first select the project in the Projects panel, then click on the Projects button. The Copy choice will then be available. Clicking on it will result in a dialog in which you can enter the name for the new project as seen in Figure 7. As before, the new project will be placed in the current Projects folder and displayed in the Projects panel. Thereafter, you can open it with Text Editor button on the main window, Figure 2, (which will now be labeled Edit Project) or simply click on it.

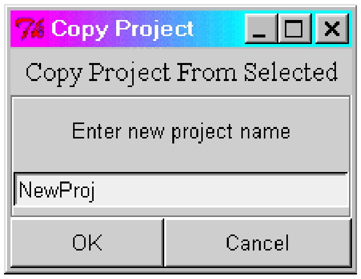


Figure 7 Copy project dialog.

8.2.2 Make Package, Make Clean, or Make CleanALL

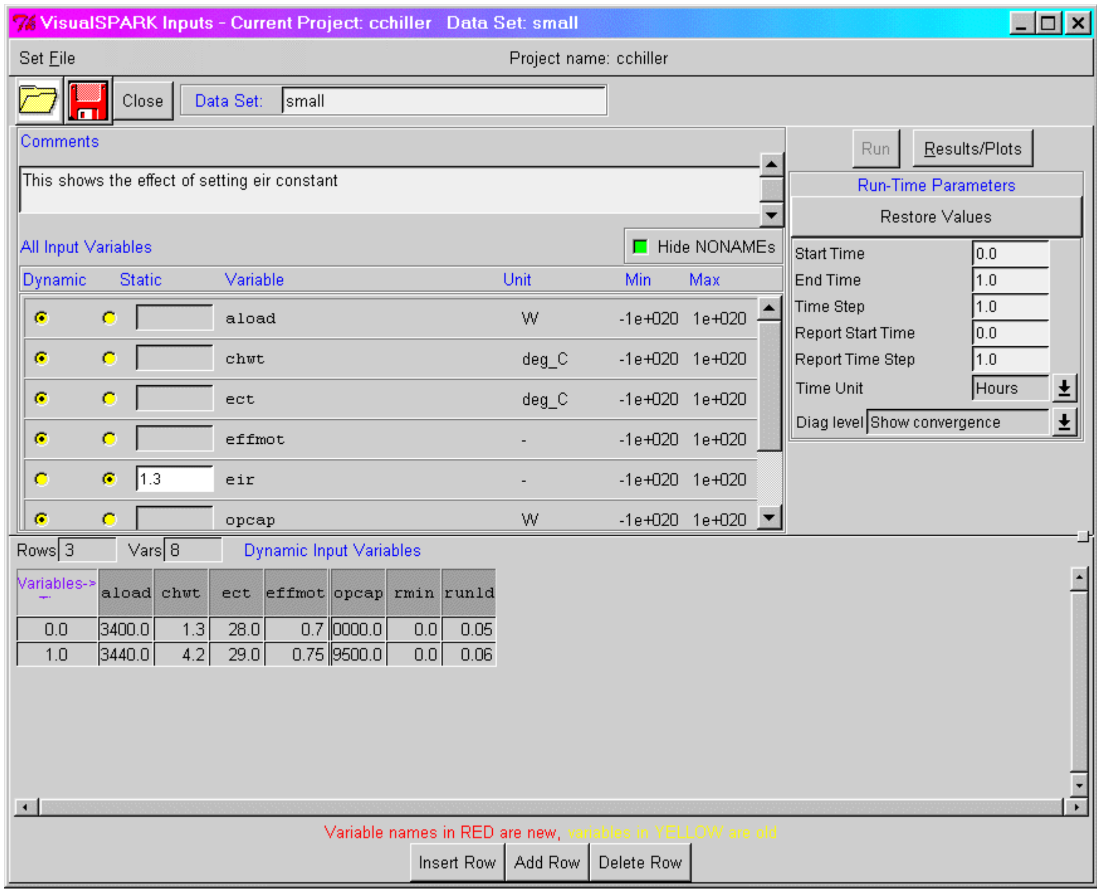
As can be seen in Figure 5, the Projects menu has three "Make" options in addition to the project creation options. These are provided for project file maintenance tasks that may need to be done occasionally. The Make Clean option removes various intermediate and results files from a project. This needs to be done sometimes because problem errors can result in incomplete builds, leaving the project in an improper state. Make Clean will allow a fresh start on the problem after you have fixed the errors. No user-created files will be deleted. Make CleanALL is similar, but does a more thorough clean up and deletes all run subdirectories created by *VisualSPARK*. The Make Package option is to allow export of a project. It copies all files related to the project, including its class files, into a new directory called *projName_pkg*. By sending this directory to a colleague, you can be sure he/she will have all necessary files to run the project.

8.3 New Input Set or Edit Input Set

After creating a new project, typically, you would then want to create an input set for it. The New Input Set button in the main *VisualSPARK* window, Figure 2, allows you to do this. Since input sets are always associated with a particular project, a project must be selected in the Projects panel before this button is available. Clicking on this button brings up the input editor window as seen in Figure 8.

If an existing input set is selected in the Projects panel instead of just a project, the Edit Input Set button will be available. Clicking it will launch the same editor as used for creating new input sets. However, in this case the fields will contain existing values.

At the top of the Input Editor window is a File menu choice, under which you will find the familiar Open, Save, Save As, and Close choices. Next there is a tool bar with the normal icons for Open, Save, and Close. To the right of the tool bar is a field showing the name of the current input set file.


VisualSPARK Inputs - Current Project: cchiller Data Set: small

Project name: cchiller

Data Set: small

Comments

This shows the effect of setting eir constant

All Input Variables

🟩 Hide NONAMEs

Dynamic	Static		Variable	Unit	Min	Max
●	○		aload	W	-1e+020	1e+020
●	○		chwt	deg_C	-1e+020	1e+020
●	○		ect	deg_C	-1e+020	1e+020
●	○		effmot	-	-1e+020	1e+020
○	●	1.3	eir	-	-1e+020	1e+020
●	○		opcap	W	-1e+020	1e+020

Run-Time Parameters

Start Time	0.0
End Time	1.0
Time Step	1.0
Report Start Time	0.0
Report Time Step	1.0
Time Unit	Hours
Diag level	Show convergence

Rows 3 Vars 8 Dynamic Input Variables

Variables->	aload	chwt	ect	effmot	opcap	rmin	runld
0.0	3400.0	1.3	28.0	0.7	0000.0	0.0	0.05
1.0	3440.0	4.2	29.0	0.75	9500.0	0.0	0.06

Variable names in RED are new, variables in YELLOW are old

Figure 8 Input editor.

8.3.1 Input Variables

Below the tool bar, the input editor has three panels. At the upper left is an area to put useful comments about the data set. Below that is a panel labeled All Input Variables listing all variables designated as *input* in the project problem file. The principal purpose of the All Input Variables panel is to allow you to classify the input variables into two categories:

- **Dynamic:** These are input variables for which you will want to give time-varying values in the lower left panel.
- **Static:** These are input variables for which you want to give constant values in the adjacent field.

You indicate the category for each variable by clicking on the "radio buttons" (small circles).

At the top of the All Input Variables panel is a check button labeled Hide NONAMEs. The purpose of this button is to allow omission of SPARK "no name" variables, i.e., those that appear only inside macro objects and having no user-assigned names. These are often quite numerous in large projects, and checking this button will avoid needless clutter in the table of variables in the input editor window.

In addition to its role in categorizing inputs, the All Input Variables panel displays other information about the variables. Shown to the right of each variable name are its units, and minimum and maximum values. These are determined from the problem definition file in the manner discussed in the User's Manual, and cannot be changed in this panel. Additionally, note that if the window size is reduced this information may not be fully visible.

If a variable name is not fully visible (e.g. because it is too long), placing the mouse over the name will popup a temporary window showing the full name.

The lower left panel is a table in which you can give values for each input variable categorized as Dynamic in the upper left panel. The first column in the table represents time, and the others represent the input variables identified as Dynamic. The first row of the table lists the names of these variables. In subsequent rows, a time value is entered in column one, followed by numerical values for the variables at that time. As explained in the User's Manual, SPARK interpolates between the given time points to get intermediate values. The table behaves in a spread-sheet like manner, so you can scroll up and down or left and right as needed. The buttons at the bottom allow you to Insert a row above a selected row, Add a row after a selected row, or Delete a selected row. The number of columns is determined automatically by the number of Dynamic input variables. If an input variable is selected as Static, its column in the table is made "invisible", because the value for a Static input variable is taken from the entry in the top panel.

8.3.2 Run-Time Parameters

The upper right panel has fields in which the Run-Time Parameters are placed. You must enter appropriate values for each of the six items. For problems to be solved only once, enter 0 for both Start Time and End Time, any nonzero value for the Time Step and Report Time Step, and 0 for the Report Start Time. For dynamic problems, you enter appropriate values determined from the mathematical model and numerical considerations. The Restore Values button at the top of the panel can be used to restore the fields to values that existed when the editor was started. The two buttons above the Run-Time Parameters fields are duplicates of buttons of the same name in the main *VisualSPARK* window.

Below the time information there is a pull-down menu from which you may choose a "diagnostic" level of output from the solver. The choices are "Silent" (no diagnostic output), "Show convergence", "Show reported variables" and "Show convergence and reported variables".

The diagnostic level "Show convergence" applies only to iterative components and shows the following information at each iteration :

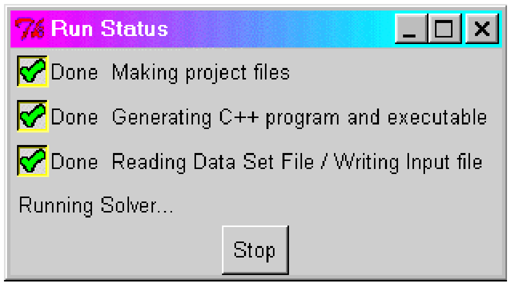
- the iteration count (0 corresponds to the prediction stage),
- the value of the Euclidean norm of all residuals comprising this component, and
- the convergence error, tolerance, value and name of the worst-offender variable.

The convergence error corresponds to the increment between successive iterations for the variable in question. If the iteration count is preceded with a "R", then the convergence error corresponds to the residual value for the inverse matched with the variable in question. The diagnostic level "Show reported variables" shows at the end of each time step the values of the problem variables for which the construct REPORT was specified in the problem. The diagnostic level "Show convergence and reported variables" combines both previous diagnostic outputs.

8.4 Running

8.4.1 The Run Command

When you have properly defined a project and an input set, select the input set in the Projects panel and click on the Run button either in the main *VisualSPARK* window, Figure 2, or in the Input Editor window, Figure 8. This builds the solver and executes the problem solution file, much as described in Section 7. A run status box is displayed as shown in Figure 9.


Run Status

✓ Done Making project files

✓ Done Generating C++ program and executable

✓ Done Reading Data Set File / Writing Input file

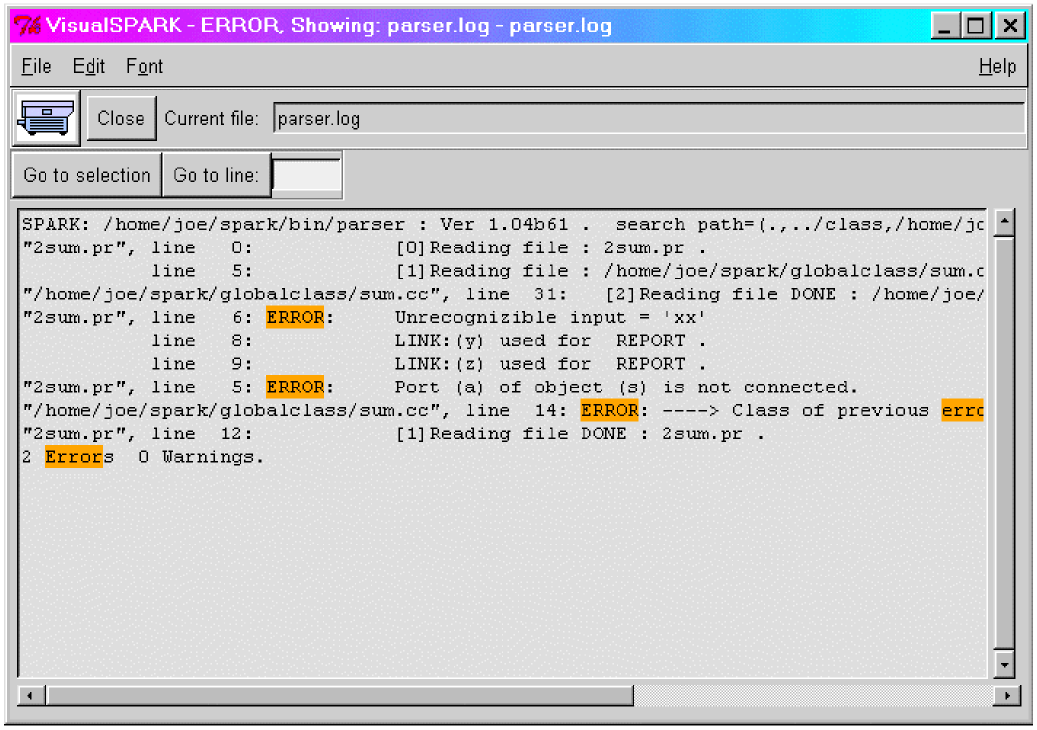
Running Solver...

Figure 9 Run status report.

The check marks are displayed as the build process progresses. Provided no errors are encountered during the build, the last stage, Running Solver, will be reached. This is the numerical solution phase. When it completes, the status report box closes. Note that while the run is in progress the Run button in the main window is highlighted in bright red. No other action can be taken with the project until the first three steps are complete and the solver is started, as indicated by the status report box text. However, the Stop button may be pressed at any time to abort the process.

8.4.2 Log Files and Error Reports

Errors can arise in any of the several steps that take place as a result of the Run command. These errors are reported in various log files as discussed in Section 7.1.4. When using the *VisualSPARK* interface, errors result in automatic opening of all log files. You should examine these carefully, beginning with *run.log* to determine where the error occurred. For example, if we deliberately introduce spurious text in the *2sum.pr* file and run it we get the *parser.log* file as shown in Figure 10.



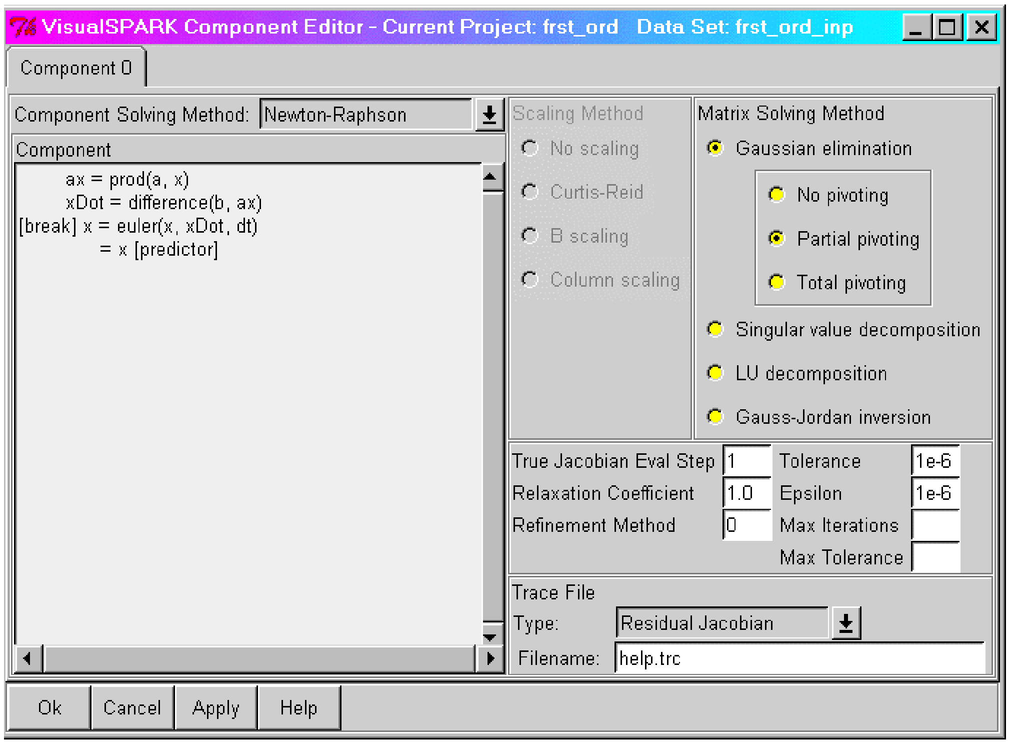
```
SPARK: /home/joe/spark/bin/parser : Ver 1.04b61 .  search path=(.,../class,/home/jc
"2sum.pr", line   0:            [0]Reading file : 2sum.pr .
           line   5:            [1]Reading file : /home/joe/spark/globalclass/sum.c
"/home/joe/spark/globalclass/sum.cc", line  31:   [2]Reading file DONE : /home/joe/
"2sum.pr", line   6: ERROR:     Unrecognizible input = 'xx'
           line   8:            LINK:(y) used for  REPORT .
           line   9:            LINK:(z) used for  REPORT .
"2sum.pr", line   5: ERROR:     Port (a) of object (s) is not connected.
"/home/joe/spark/globalclass/sum.cc", line  14: ERROR: ----> Class of previous errc
"2sum.pr", line  12:            [1]Reading file DONE : 2sum.pr .
2 Errors  0 Warnings.
```

Figure 10 Project parser log.

8.5 Component Preferences

The Preferences button in the main window, Figure 2, brings up the Component Editor that allows you to specify advanced solution controls such as choice of solution methods[9] and solution accuracy. This window is shown in Figure 11.

[9] The initial release may not have all solution methods active.


VisualSPARK Component Editor - Current Project: frst_ord Data Set: frst_ord_inp

Component 0

Component Solving Method: Newton-Raphson

Component

```
      ax = prod(a, x)
      xDot = difference(b, ax)
[break] x = euler(x, xDot, dt)
         = x [predictor]
```

Scaling Method: No scaling, Curtis-Reid, B scaling, Column scaling

Matrix Solving Method: [✓ Gaussian elimination] (No pivoting, [✓ Partial pivoting], Total pivoting), Singular value decomposition, LU decomposition, Gauss-Jordan inversion

True Jacobian Eval Step	1	Tolerance	1e-6
Relaxation Coefficient	1.0	Epsilon	1e-6
Refinement Method	0	Max Iterations	
		Max Tolerance	

Trace File
Type: Residual Jacobian
Filename: help.trc

Figure 11 Component Editor window.

In many instances, the default solution method and accuracy controls will suffice, so you often do not have to change anything. However, if the solution fails, you may want to visit this dialog. Tolerance may be thought of as the maximum acceptable absolute error. Iterative solution stops when the absolute difference between two successive values of all iteration variables falls below this value. Epsilon is the step size on the iteration variables used to calculate numerical derivatives. The User's Manual should be consulted for more details on choices available here.

It is important to note here that SPARK automatically decomposes your problem into separately solvable pieces, called *components*[10]. Each component can have its own solution method and accuracy specifications. Thus the Component Editor dialog is tabbed across the top. By clicking on a particular tab you get a form for setting the controls for that component. Since the problem considered here has a single component, Figure 11 has a single tab.

Note the panel in the lower left portion of the dialog. This shows the equation sequence for the component, as found in the equations file. This may offer some guidance in selection of solving method for the component.

SPARK can optionally capture intermediate, diagnostic output on a component by component basis. There are two fields in the lower right corner of the Component Editor window for requesting this tracing mechanism. A pull down menu labeled Type gives the option of writing diagnostics for: All variable values, Residuals, the Jacobian, Increments or None, to the file designated in the Filename field. The User's Manual should be consulted for more details on each tracing mechanism.

Finally, the menu bar at the bottom of the Component Editor window presents buttons to allow you to accept or reject changes made in the editor, or to seek Help. The OK button accepts all changes and leaves the Component Editor, while Apply accepts them but stays in the editor so you can make changes for other

[10] The term "Component" is used here to mean a *strongly connected component* of the problem graph, not models of physical components.

components. Cancel discards any changes you may have made to any of the components and leaves the editor.

8.6 Viewing and Plotting Results

To view the output, just click on the Results/Plot button in the main window, Figure 2, or in the Input Editor window, Figure 8. This brings up the menu shown in Figure 12.

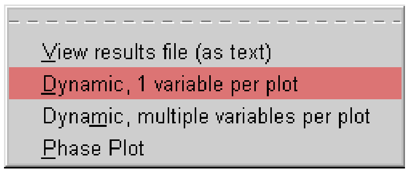

View results file (as text)
Dynamic, 1 variable per plot
Dynamic, multiple variables per plot
Phase Plot

Figure 12 Results/Plot menu.

The top-most choice will display the results as a text file in an edit window. Results for simple, algebraic problems are best viewed in this manner. The User's Manual provides an explanation of the contents and format of this data. All choices present a popup dialog to allow you to choose the file to plot. This may either be the normal output file or the output from the trace file

The next three menu choices offer different ways of plotting the results for dynamic problems. The "Dynamic, 1 variable per plot" choice presents the results as a sequence of plots, each with a single reported variable plotted versus time. When this button is clicked the window shown in Figure 13 appears.

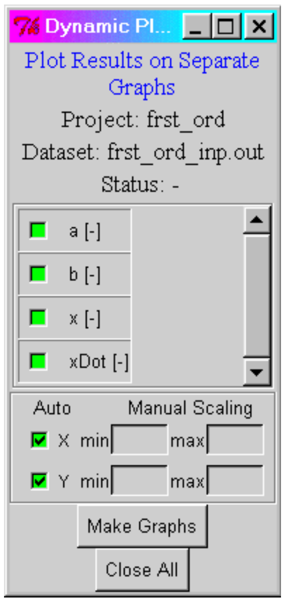


Figure 13 Single variable plot dialog.

All problem variables with a *report* designation in the problem file (see SPARK User's Manual) appear in the upper panel with adjacent square selection symbols. You can select variables to be plotted by clicking on the squares, causing them to become highlighted. Automatic scaling is done for X and/or Y axes[11] if so indicated with a highlighted box. If an axis is not checked, you must give minimum and maximum values for scaling the axis. When Make Graphs is clicked, a plot for each selected variable will be generated as seen in Figure 14. The Close All button will close all generated plots. Otherwise, you can close them one at a time with the Close button on each plot. The "Print" button sends a plot to the default printer.

[11] Note that the X and Y axes in the dialog refer to the plot horizontal and vertical axes respectively, not to the problem variable names.

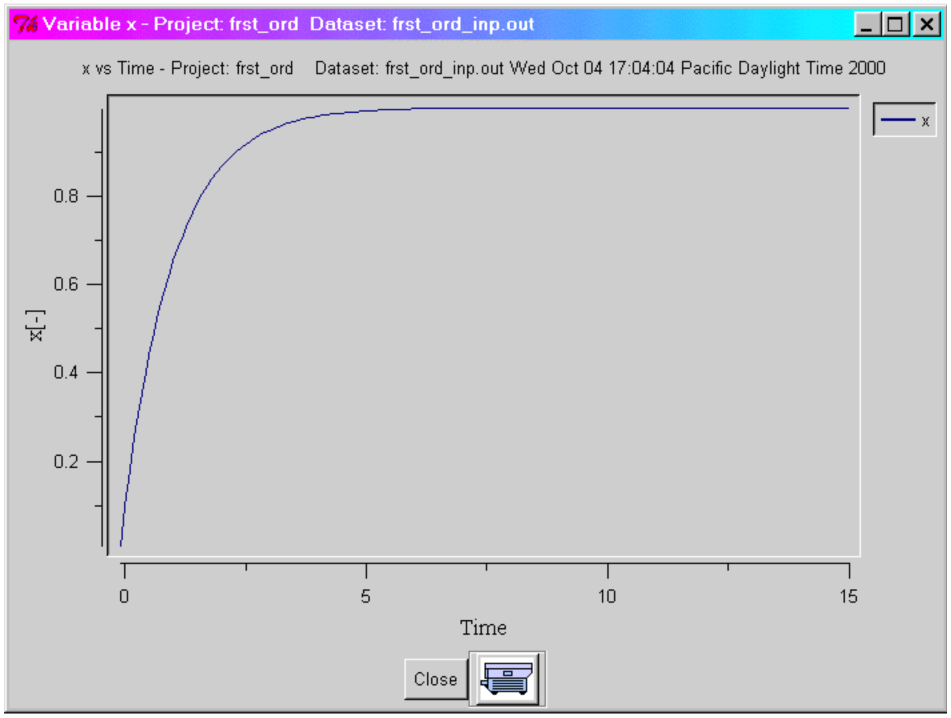

Variable x - Project: frst_ord Dataset: frst_ord_inp.out
x vs Time - Project: frst_ord   Dataset: frst_ord_inp.out Wed Oct 04 17:04:04 Pacific Daylight Time 2000
x[-]
0.2
0.4
0.6
0.8
0
5
10
15
Time
x
Close

Figure 14 Dynamic plot of single variable.

The Dynamic, multiple variables per plot option in Figure 12 produces a somewhat different dialog, Figure 15.

As before, variables to be plotted are selected by clicking on the adjacent squares in the top panel. However, as can be seen, you can designate each variable to be plotted on either a left (Y1) or right (Y2) axis. Automatic scaling is done for X and/or Y axes if so indicated with a highlighted box. If an axis is not checked, you must give minimum and maximum values for scaling the axis. Then when the Make button is clicked, a graph with all selected variables will be displayed to the right of the dialog, as shown in Figure 16.

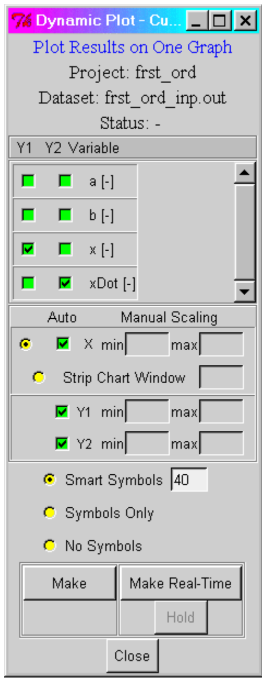


Figure 15 Multiple plots per graph.

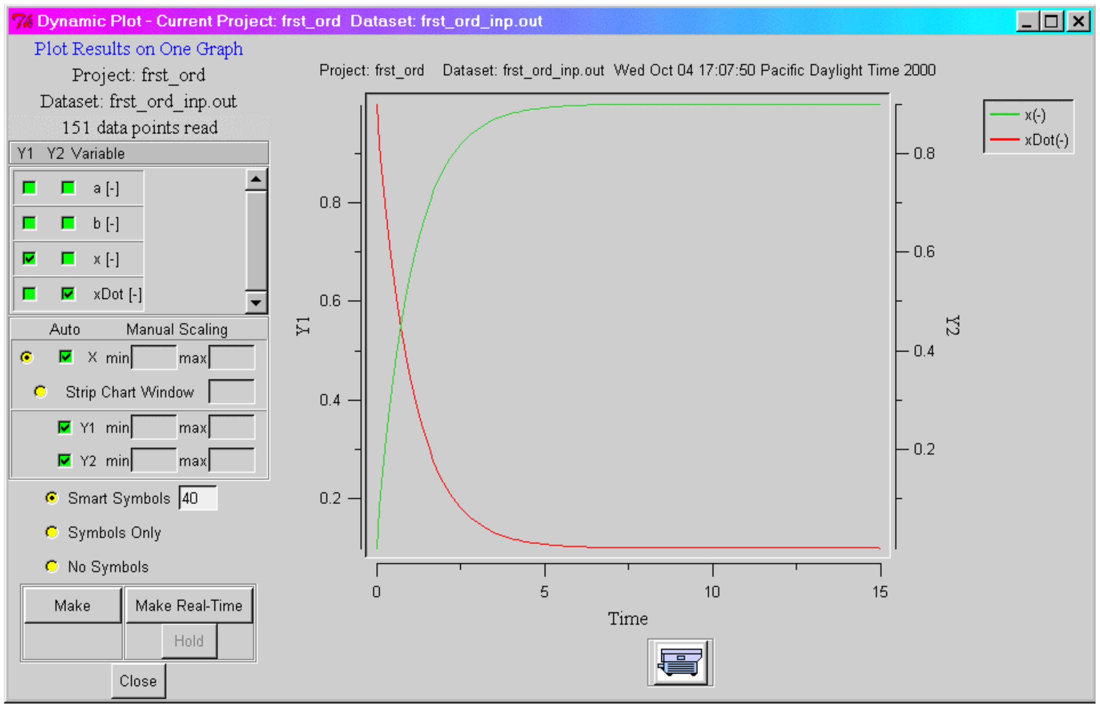


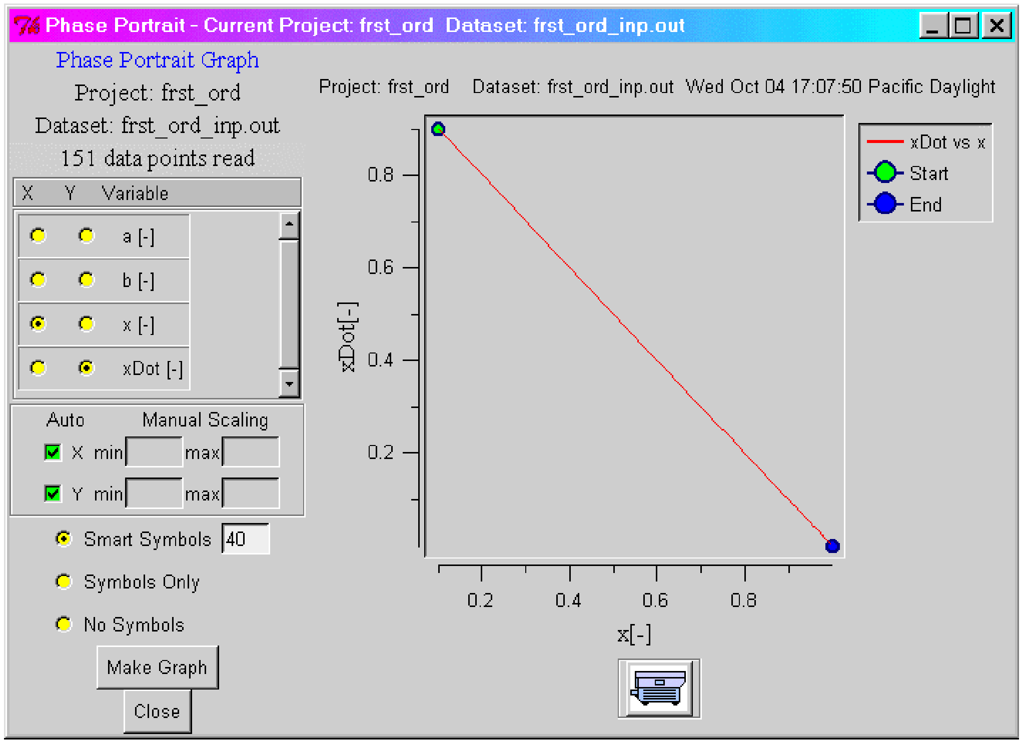
Figure 16 Multi-variable plot graph.

As requested in this example, the *x* problem variable is plotted against an ordinate scaled to the left vertical axis, while *xDot* relates to the right vertical axis. Here, we have accepted the default automatic scaling. Alternatively, you can set the scales manually for all axes. The Smart symbols option means that plotting symbols are used when appropriate, but if the graph gets too crowded, as would be the case in this example, they are omitted. The field adjacent to the Smart Symbols option allows you to specify the maximum number of plotted points before plot symbols are omitted. The Print button will display a dialog for printing the graph to an installed Windows printer.

Note that there is a Make Real-Time button in addition to the Make button. If the simulation runs for a long time, the Make Real-Time option allows you to see the graph developing on the screen as the solution proceeds. There are two options for the real-time plot display. When the Strip Chart Window diamond is checked, a fixed-scale, moving time axis is used, so the results appear as they would on a strip chart recorder. Otherwise, the time axis is rescaled dynamically as time advances. To use the Make Real Time feature meaningfully you have to execute the “Results / Plots | Dynamic, multiple variables per plot | Make-Real-Time” sequence *while the run is in progress*, i.e., before the Run Status window, Figure 9, has closed. Otherwise, the Make Real-Time button will produce a plot the same as with Make.

The Hold button just below the Make Real-Time button allows you to temporarily freeze the graph to examine it more closely. The label then changes to Resume, and pressing it again will allow the graph to resume updating as the solver produces more output.

The last plotting choice, Phase Plot, on the Results/Plot menu, Figure 12, is for plotting selected variables against each other, rather than against time. This option provides the dialog allowing you to select the plot variables, and when the Make Graph button is selected the phase plot is displayed to the right of the dialog. Figure 17 shows a plot of *x* vs. *xDot* for the *frst_ord* example.



X: x [-]; Y: xDot [-]
Auto: ✓ X, ✓ Y
[✓ Smart Symbols] 40

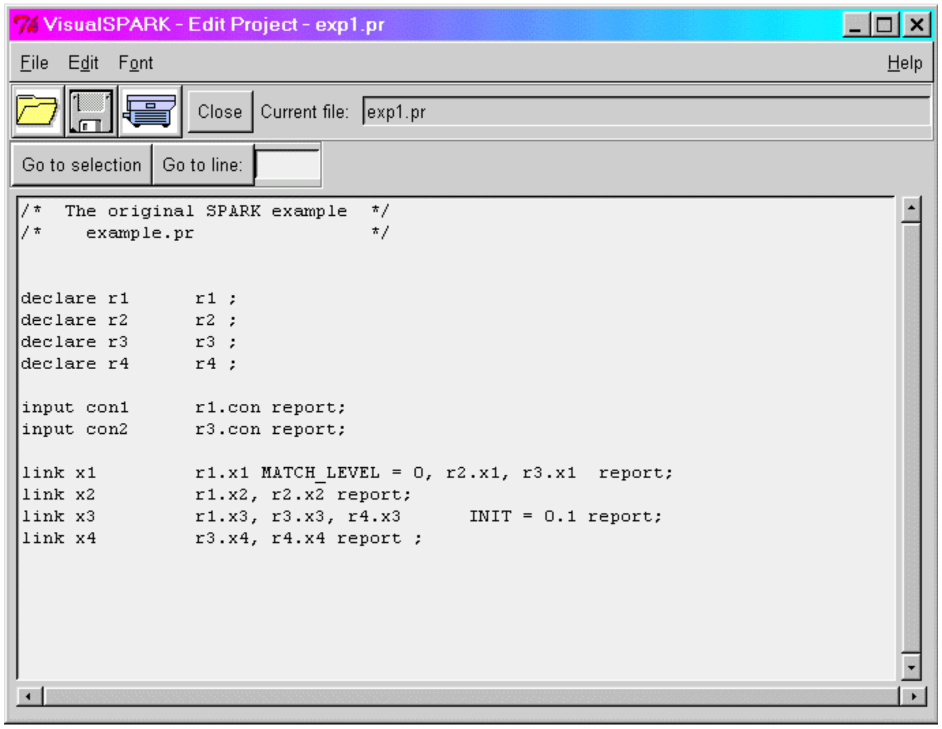
Figure 17 Phase plot.

For all of the graph types, clicking the left mouse button in the graph area and dragging out a rectangle will zoom that area to fill the space. One may zoom in multiple times. Clicking the right mouse button will un-zoom one level at a time.

Note that the plot windows shown in Figure 14, Figure 16, and Figure 17 have been reduced in size to fit in this Guide. On the screen, they are scaled more generously, making them more readable. As with most windows on your desktop, they can be scaled dynamically to make them larger or smaller.

8.7 Editing Projects and Classes

Either projects (i.e., problem files) or classes can be edited within the *VisualSPARK* interface. There are two ways of opening existing projects or classes. The easiest way is to double click on the desired project or class in the appropriate panel in the main *VisualSPARK* window, Figure 2. Alternatively, you can select it in the panel and then click on the Edit button in the upper menu bar in the main window. Either method will open the associated source file in an edit window, Figure 18.



```
/*  The original SPARK example  */
/*    example.pr                */


declare r1       r1 ;
declare r2       r2 ;
declare r3       r3 ;
declare r4       r4 ;

input con1       r1.con report;
input con2       r3.con report;

link x1          r1.x1 MATCH_LEVEL = 0, r2.x1, r3.x1  report;
link x2          r1.x2, r2.x2 report;
link x3          r1.x3, r3.x3, r4.x3      INIT = 0.1 report;
link x4          r3.x4, r4.x4 report ;
```

Figure 18 Editing projects or classes.

Once opened with the editor, you can make changes as needed. The User's Manual should be consulted for information on SPARK syntax.

The editor offers the basic functionality most users will be familiar with, so its operation will not be discussed at length. As can be seen, there are icons for opening, saving, printing, and closing of the file. These functions are also available under the File menu. The Edit menu offers Undo, Search, and Replace options. The Font menu allows selection of a range of font sizes.

You can maneuver the insertion point with the mouse, using side and bottom scroll bars if needed. Also, there are two Go To buttons for going to specific lines in the file. The Go To Line button will go to the line whose number is entered in the adjacent field. The Go To Selection button also goes to a line number, but in this case the target line number is identified through selection. Typically the selected number will be in a different window. For example, an error log file such as shown in Figure 10 has line numbers indicating the offending line in the source file. If you select one of these line numbers in the error log file by double clicking, then open the source file for editing and click on the Go To Selection button, you will be taken to the line that needs attention.

Note that you must save the file before you run the problem or your changes will not take effect. **Always save any SPARK file after changing it.**

8.8 Creating SPARK Classes

At the bottom of the Classes panel in the main window, Figure 2, there is a button labeled New Class. Pressing this button brings up several options offering assistance in creation of new SPARK classes, Figure 19.

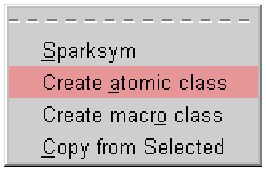


Figure 19 New Class menu.

The first choice (*Sparksym*) offers symbolic algebra tools for automatic creation of atomic classes, macro classes, and SPARK problem files. It pops up a panel into which you may enter an equation to be solved by one of several symbolic algebra tools, such as Mathomatic,[12] Maple, Mathematica or MACSYMA, that you can use to develop new SPARK atomic classes. This selection first brings up a dialog for entry of a name for the new class, Figure 20.

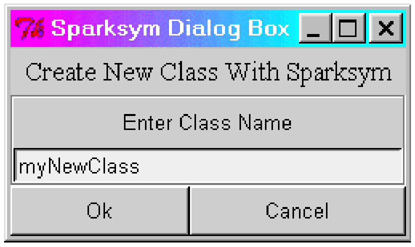


Figure 20 Sparksym dialog.

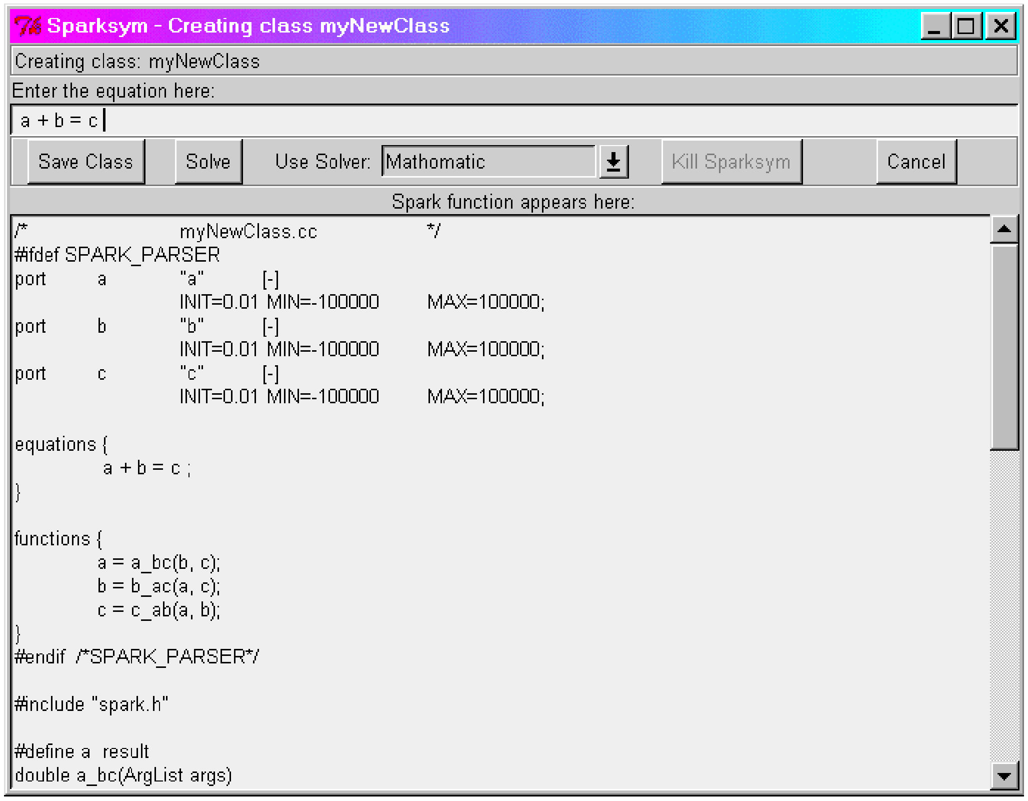
After entering a name and pressing OK, a *Sparksym* solve window is opened, Figure 21.

The line at the top is for input of an algebraic equation of the form <*expression*> = < *expression*>. Then when the Solve button is pressed a complete SPARK atomic class is generated and displayed in the lower panel. When you press Save Class, the class gets saved under the specified class name with a *.cc* extension in the active class directory. Pressing Close without first saving will result with a confirmation dialog.

In generating the atomic class, *Sparksym* calls the chosen symbolic algebra solver to attempt to generate explicit inverse functions for every variable in the expression. If explicit inverses are not found for some variables, "implicit inverses" are used, as explained in the User's Manual. These inverses are embedded as functions in the generated class, using names generated from the variable names. Also, every variable is placed in the class interface using a *port* statement with nominal values for *init*, *min* and *max* values, and a *units* string of [-], i.e., unspecified units.

Although the class is directly usable in many cases, you may want to open it with the class editor and customize it for your purposes. Additionally, you should always examine the functions carefully before use since computer algebra sometimes gives unexpected results.

[12] *Sparksym* includes a licensed derivative of the Mathomatic symbolic algebra tool. The full DOS shareware version of Mathomatic is available from *www.lightlink.com/george2*.


Sparksym - Creating class myNewClass
Creating class: myNewClass
Enter the equation here:
Save Class
Solve
Kill Sparksym
Cancel
Spark function appears here:

Use Solver: Mathomatic

```
a + b = c
```

```
/*              myNewClass.cc           */
#ifdef SPARK_PARSER
port    a       "a"     [-]
                INIT=0.01 MIN=-100000   MAX=100000;
port    b       "b"     [-]
                INIT=0.01 MIN=-100000   MAX=100000;
port    c       "c"     [-]
                INIT=0.01 MIN=-100000   MAX=100000;

equations {
        a + b = c ;
}

functions {
        a = a_bc(b, c);
        b = b_ac(a, c);
        c = c_ab(a, b);
}
#endif /*SPARK_PARSER*/

#include "spark.h"

#define a  result
double a_bc(ArgList args)
```

Figure 21 Sparksym solve window.

The symbolic algebra solver that comes bundled with *VisualSPARK* is Mathomatic. Although not as complete as larger computer algebra systems, Mathomatic may meet many of your needs for SPARK class development. Its primary limitation is that only standard binary operations such as +, -, *, /, and ^, and unary minus -, are recognized. If you find that Mathomatic does not meet your needs, you may want to install MACSYMA, Maple, or Mathematica, more capable symbolic algebra tools. Once you have any of these installed, you may choose them from the pull-down menu labeled “Use Solver”. Like Mathomatic, these options allow you to automatically create SPARK atomic classes. However, because the MACSYMA, Maple and Mathematica programs are more robust, they can generate many inverses for which Mathomatic fails. Additionally, macro classes and entire SPARK problems can be generated from given equations. The complexities of these tools prevent explanation here.

The last three options on the New Class menu, Figure 19, are for creation of new classes without use of symbolic tools. The Copy from Selected option prompts for a new class name and then creates a copy of the class currently selected in the Class panel, placing it in the active class directory. You should then edit the new class as needed, as discussed in Section 8.7. The two Create choices prompt for a new class name and then open an edit window with skeleton classes of the indicated type, atomic or macro. These can be edited to finish creation of the new class of the desired properties.

9 Support

Direct *VisualSPARK* downloading and installation questions to

sparksupport@SimulationResearch.lbl.gov

Direct questions on using *VisualSPARK* to Edward F. Sowell of Ayres Sowell Associates, Inc. at

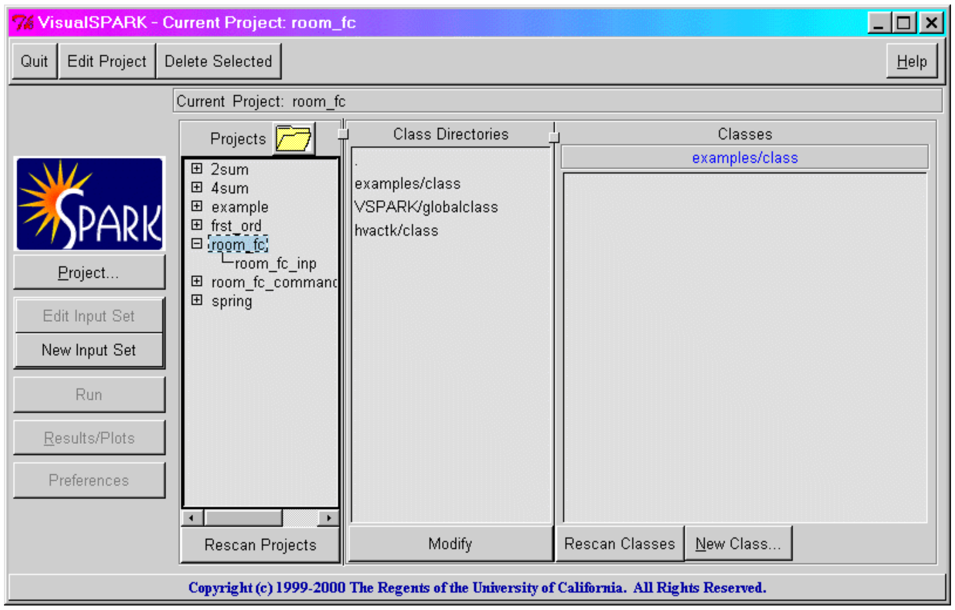
sowell@fullerton.edu

For updates, see http://SimulationResearch.lbl.gov/VisualSPARK

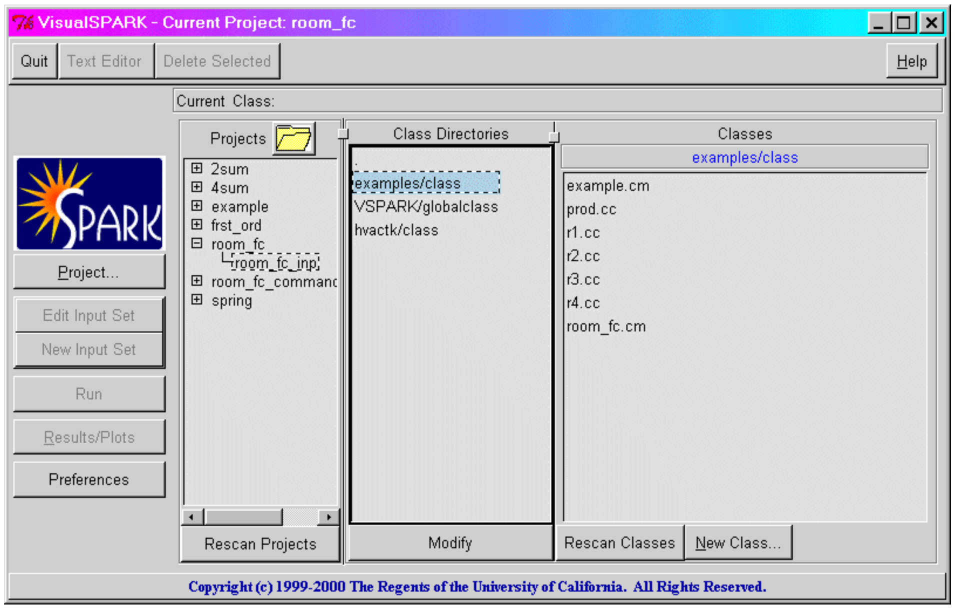
Appendix A Tutorial

A.1 Getting Started

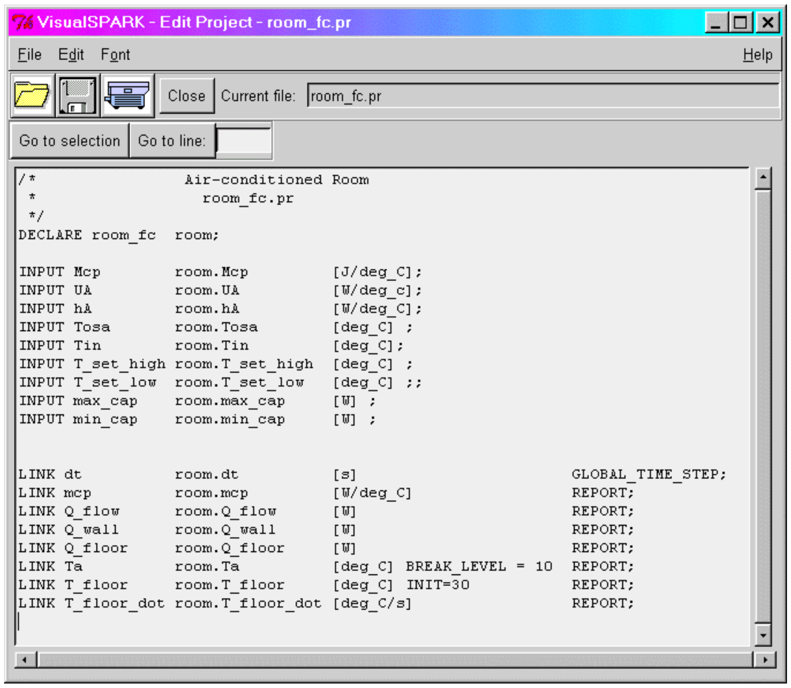
Let's try running the *room_fc* (air conditioned room) example from the *doc/examples* directory that comes with *VisualSPARK*. This is fully described in section 2.7 of the *SPARK User's Manual*. From the following screenshot we see it is one of the seven example projects. After clicking on the "+" sign to the left of the name, it opens to show that there is a dataset or set-file called *room_fc_inp*. This contains input data along with the run-time information needed to run the model:



Notice that when we clicked on the "+" sign to open the project, the class directories that this project uses appeared in the panel immediately to the right. If you click on one of those directory names, the class files in that directory will appear in the panel to its right. Note that not all class files are necessarily used by the project, just that they are available for its use:



Now let's look at the SPARK code for the model itself. Reselect the project by clicking on the "*room_fc*" label and notice that the second button in the top of the *VisualSPARK* panel changes from "Text Editor" to "Edit Project", signifying that if it is pressed, the editor will start with the project's .pr (problem) file. Now press it to see the following screen:

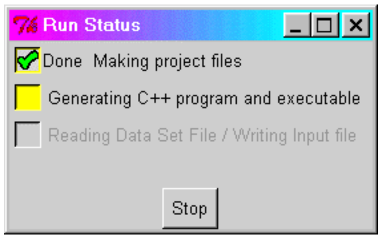


```
/*                  Air-conditioned Room
 *                    room_fc.pr
 */
DECLARE room_fc  room;

INPUT Mcp         room.Mcp          [J/deg_C];
INPUT UA          room.UA           [W/deg_c];
INPUT hA          room.hA           [W/deg_C];
INPUT Tosa        room.Tosa         [deg_C] ;
INPUT Tin         room.Tin          [deg_C];
INPUT T_set_high  room.T_set_high   [deg_C] ;
INPUT T_set_low   room.T_set_low    [deg_C] ;;
INPUT max_cap     room.max_cap      [W] ;
INPUT min_cap     room.min_cap      [W] ;


LINK dt           room.dt           [s]                          GLOBAL_TIME_STEP;
LINK mcp          room.mcp          [W/deg_C]                    REPORT;
LINK Q_flow       room.Q_flow       [W]                          REPORT;
LINK Q_wall       room.Q_wall       [W]                          REPORT;
LINK Q_floor      room.Q_floor      [W]                          REPORT;
LINK Ta           room.Ta           [deg_C] BREAK_LEVEL = 10     REPORT;
LINK T_floor      room.T_floor      [deg_C] INIT=30              REPORT;
LINK T_floor_dot  room.T_floor_dot  [deg_C/s]                    REPORT;
```

Here, we could make changes to the problem, which is written in the SPARK language. For now, close the Edit Project window.

A.1.1 Running the Model

Let's go ahead and use the room_fc model as is. Click on the dataset called "room_fc_inp" under the project label "room_fc" in the main panel. This selects the dataset we would like to run the model with. Next, press the "Run" button on the main panel and an information window will pop up showing the progress of building, compiling and running the model:



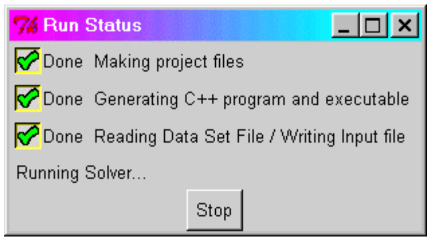
Done Making project files
Generating C++ program and executable
Reading Data Set File / Writing Input file

There are four steps:

1. Assembling the relevant files into a C++ program
2. Compiling the C++ program and linking with the solver library
3. Reading the dataset and creating the input file for the solver
4. Running the solver

The yellow color indicates the current step and a checkmark will appear when the step is complete. At any step you may stop the process by pressing the "Stop" button in the panel.

In the final step you will see a message that it is running the solver:



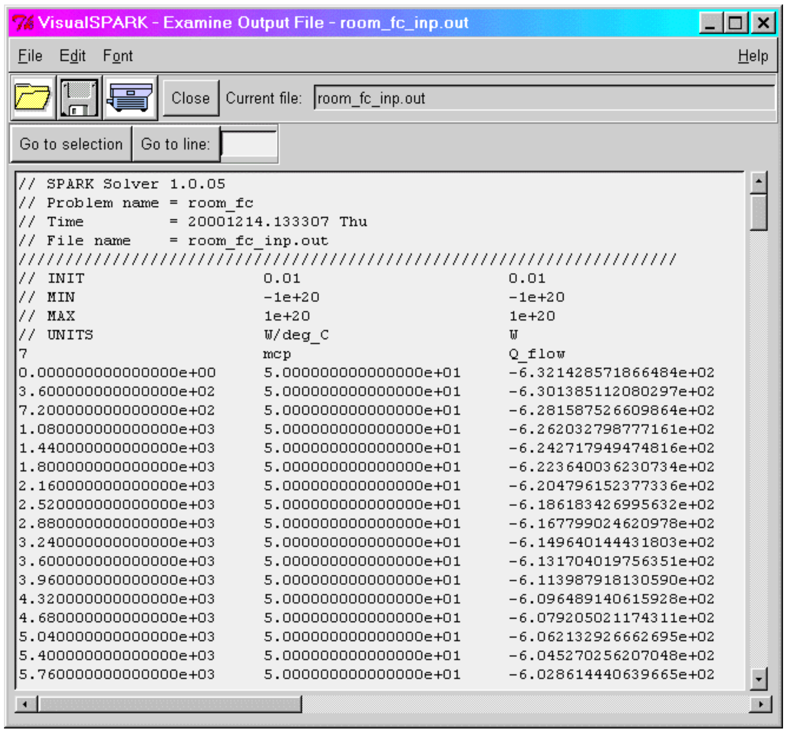
Done Making project files
Done Generating C++ program and executable
Done Reading Data Set File / Writing Input file
Running Solver...

At this point, the solver may be aborted by pressing the "Stop" button.

A.2 Viewing the Results

A.2.1 As Text

After a run (whether successful or unsuccessful), you may examine the results data either as a table of numbers or a graph. The former is achieved by choosing "View results file (as text)" from the "Results/Plots" menu in the main panel. Here is our room_fc example:



Current file: room_fc_inp.out

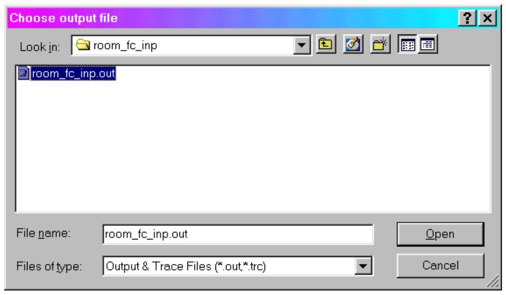
```
// SPARK Solver 1.0.05
// Problem name = room_fc
// Time         = 20001214.133307 Thu
// File name    = room_fc_inp.out
/////////////////////////////////////////////////////////////////////
// INIT                    0.01                     0.01
// MIN                     -1e+20                   -1e+20
// MAX                     1e+20                    1e+20
// UNITS                   W/deg_C                  W
7                          mcp                      Q_flow
0.000000000000000e+00      5.000000000000000e+01    -6.321428571866484e+02
3.600000000000000e+02      5.000000000000000e+01    -6.301385112080297e+02
7.200000000000000e+02      5.000000000000000e+01    -6.281587526609864e+02
1.080000000000000e+03      5.000000000000000e+01    -6.262032798777161e+02
1.440000000000000e+03      5.000000000000000e+01    -6.242717949474816e+02
1.800000000000000e+03      5.000000000000000e+01    -6.223640036230734e+02
2.160000000000000e+03      5.000000000000000e+01    -6.204796152377336e+02
2.520000000000000e+03      5.000000000000000e+01    -6.186183426995632e+02
2.880000000000000e+03      5.000000000000000e+01    -6.167799024620978e+02
3.240000000000000e+03      5.000000000000000e+01    -6.149640144431803e+02
3.600000000000000e+03      5.000000000000000e+01    -6.131704019756351e+02
3.960000000000000e+03      5.000000000000000e+01    -6.113987918130590e+02
4.320000000000000e+03      5.000000000000000e+01    -6.096489140615928e+02
4.680000000000000e+03      5.000000000000000e+01    -6.079205021174311e+02
5.040000000000000e+03      5.000000000000000e+01    -6.062132926662695e+02
5.400000000000000e+03      5.000000000000000e+01    -6.045270256207048e+02
5.760000000000000e+03      5.000000000000000e+01    -6.028614440639665e+02
```

A.2.2 Graphing The Results

The text view is not very exciting, and it is difficult to see any trends in the data. Let's look at a graph of the data instead. There are two major types of graphs – "Dynamic plot" and "Phase Plot". The first is simply a graph of the output variables versus time. The second allows you to plot one variable (X) versus another (Y) to the correlation between variables as a function of time.

A.2.2.1 The Basic Dynamic Graph

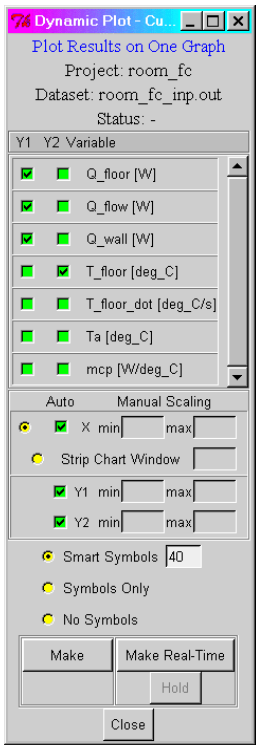
In the Dynamic graphs, you may either have a different graph for each variable to be plotted (labeled "1 variable per plot" in the menu) or plot multiple variables on either of two Y axes (labeled "Dynamic, multiple variables per plot"). Lets make a graph of multiple variables. After choosing the "Dynamic multiple variables per plot" from the "Results/Plots" menu, we choose the output file, called "room_fc_inp.out" from the dialog that appears:



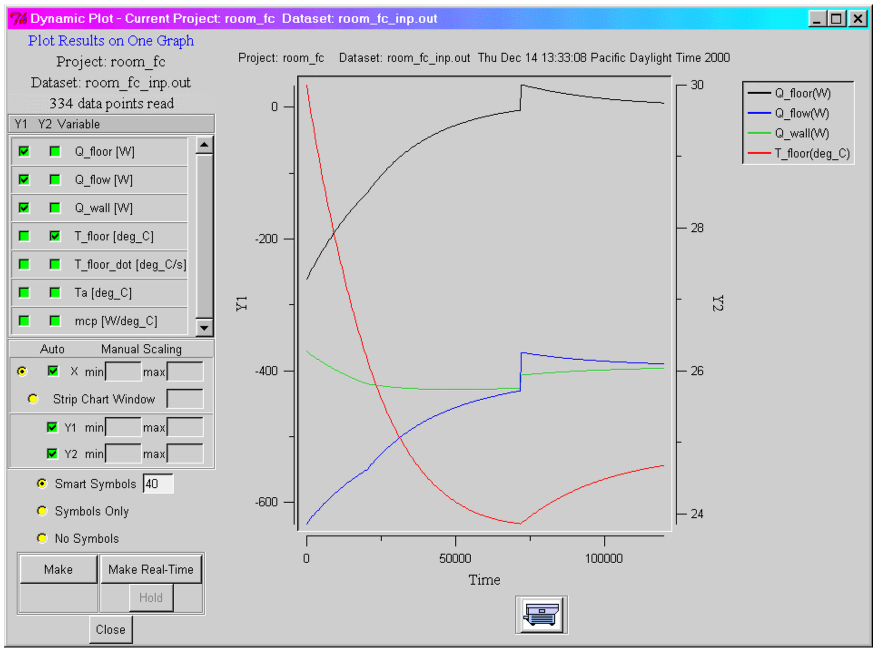
File name: room_fc_inp.out

Files of type: Output & Trace Files (*.out;*.trc)

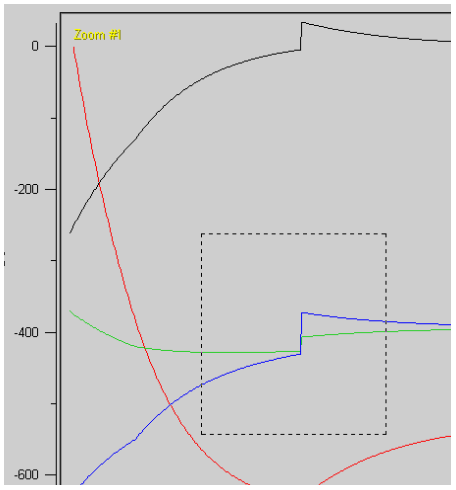
After clicking on the output file name and clicking "Open" you are presented with a panel allowing you to choose which variables to plot, and on which Y-axis you want it. Simply click on the green box of each variable you wish to see plotted:



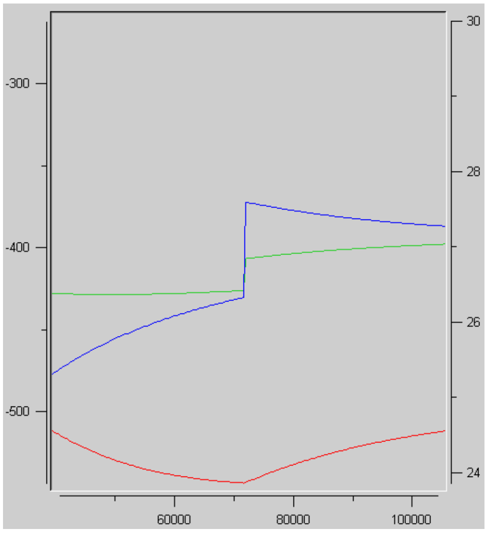
For now, we will leave the other settings at their default. These allow you to manually scale the axes, show a "window" or "strip chart" of the data in real time as the solver runs, and control the appearance of symbols on the curves. Now that you've chosen the variables, press the "Make" button to create the graph:



Now zoom in on the center of the graph where the curves make a "step". Click the mouse on one corner of a rectangle defining the area of interest, drag the mouse to the other corner and click again. This will zoom the area you outlined:



You can see the "Zoom #1" message and the rectangle. After the second button press we see the area we are interested in:



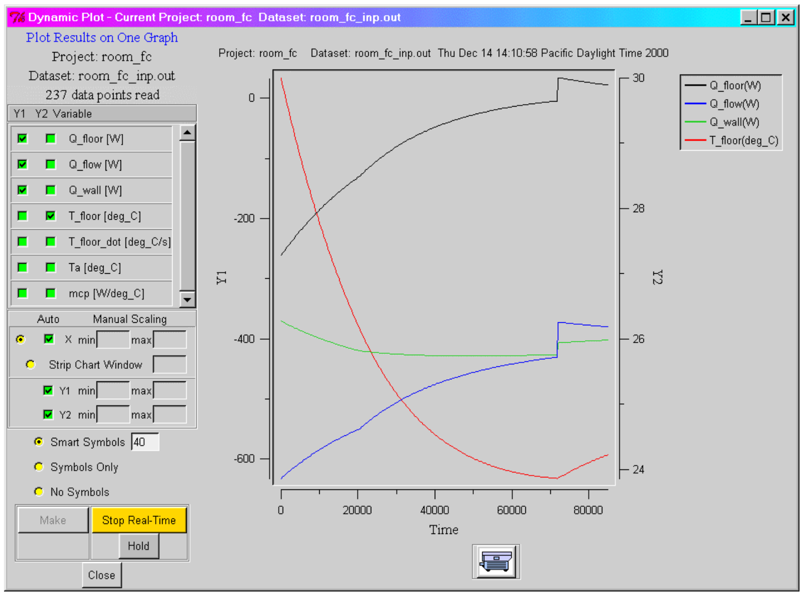
You may zoom in repeatedly for many levels. To unzoom, press the right mouse button once for each zoom level.

There is a printer icon at the bottom of the graph, which will let you print your graph. Under Windows, it will pop up the printer chooser dialog. Under Unix (Linux, Solaris, etc) it will print to the default printer defined in the PRINTER environment variable, or the printer named in the entry next to the printer icon, if it is not blank.

A.2.2.2 Real-Time Graph

Now that you have the graph displayed we can try the real-time graphing feature of *VisualSPARK*. When a simulation runs for a long time, you may wish to see results before it is finished. This allows you to stop the simulation if it is not running correctly, or simply to check its progress. At the bottom of the graph panel there is a button labeled "Make Real-Time". Press it and then press the "Run" button in the main panel and watch the curves change as the data is written to the output file. At any time during the run of the model you may press the "Hold" button to freeze screen updates of the curves. At this point the label changes to "Resume", signifying that you may press it to resume screen updates. Notice that after pressing the "Make

Real-Time” button its label changed to “Stop Real-Time. This indicates that you may stop the graph by pressing it. Here is a screenshot of the graph in progress:

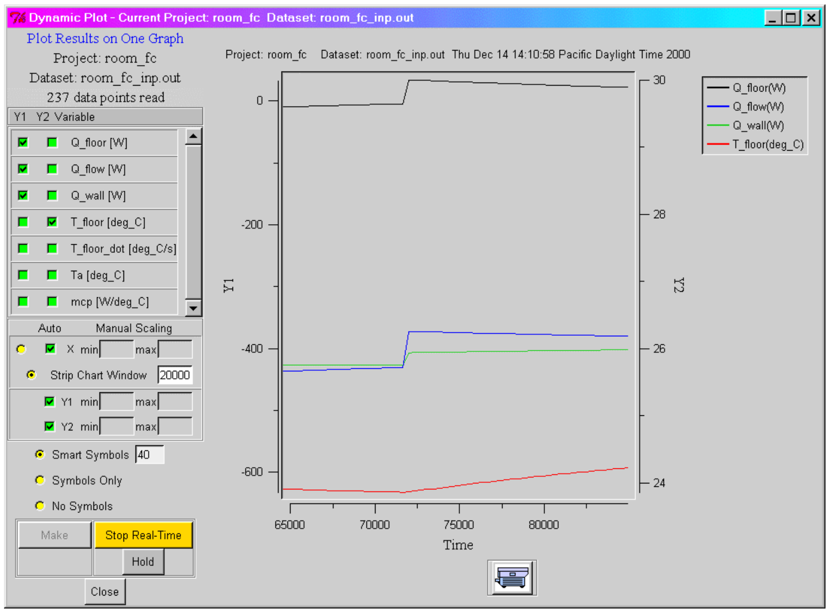


Project: room_fc
Dataset: room_fc_inp.out
237 data points read
Smart Symbols 40
Project: room_fc Dataset: room_fc_inp.out Thu Dec 14 14:10:58 Pacific Daylight Time 2000

You may have noticed the symbols appear on the curves for a time and then disappear. This is caused by the “smart symbols” feature. There are several radio buttons above the buttons to create the graph, which let you control when symbols appear on the curves. The “smart symbols” radio button tells the graphing program to put symbols on the solid line only when the number of symbols is no greater than the number in the box to the right. When the number of points exceeds that value (in our case 40) then only a solid line is drawn. The radio button below that, labeled “Symbols Only” means to plot symbols on the curves but not connect them with lines. The last radio button “No Symbols”, means to not show any symbols, only a solid line.

A.2.2.3 Strip Chart Graph

Just above the symbol control section is the scaling section. In it you may allow automatic scaling of the axes (the default) or choose your own minimum and maximum values for the X-, Y1- and Y2-axes. Notice the radio button labeled “Strip Chart Window”. During the creation of a real-time graph you may only want to look at the latest period of data. After pressing this radio button, enter the “window size” in the entry to the right. This is the amount of time in units of X values to show in the window. The example below shows the latest 20,000 time units of data:


Dynamic Plot - Current Project: room_fc Dataset: room_fc_inp.out

Plot Results on One Graph
Project: room_fc
Dataset: room_fc_inp.out
237 data points read

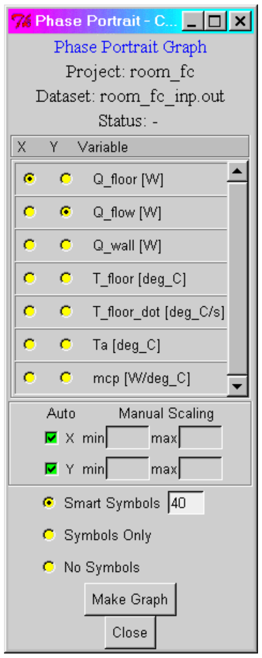
Y1	Y2	Variable
✓		Q_floor [W]
✓		Q_flow [W]
✓		Q_wall [W]
	✓	T_floor [deg_C]
		T_floor_dot [deg_C/s]
		Ta [deg_C]
		mcp [W/deg_C]

Auto / Manual Scaling
✓ X min max
● Strip Chart Window 20000
✓ Y1 min max
✓ Y2 min max
● Smart Symbols 40
Symbols Only
No Symbols

Project: room_fc Dataset: room_fc_inp.out Thu Dec 14 14:10:58 Pacific Daylight Time 2000

A.2.2.4 Phase Plot

The last graph type is the phase plot. With it you can see the relation between any two variables, as one is graphed versus the other for each time point. Let’s take a look at the relation between Q_floor and T_floor. Choose the “Phase Plot” option from the “Results/Plots” menu and choose the output file as before. You will see this panel:

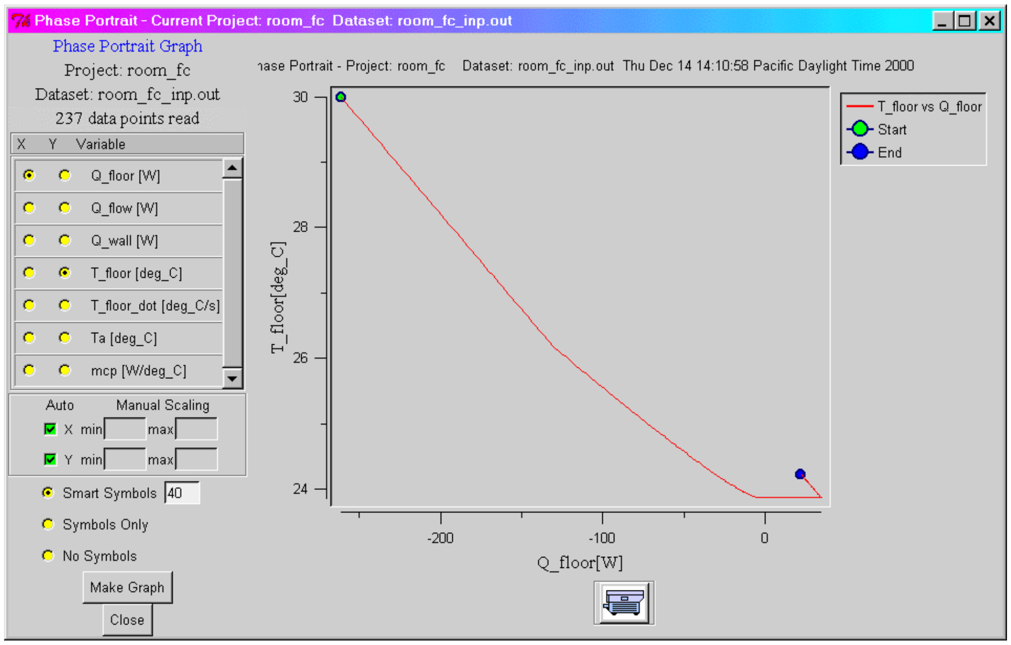

Phase Portrait - C...

Phase Portrait Graph
Project: room_fc
Dataset: room_fc_inp.out
Status: -

X	Y	Variable
●		Q_floor [W]
	●	Q_flow [W]
		Q_wall [W]
		T_floor [deg_C]
		T_floor_dot [deg_C/s]
		Ta [deg_C]
		mcp [W/deg_C]

Auto / Manual Scaling
✓ X min max
✓ Y min max
● Smart Symbols 40
Symbols Only
No Symbols

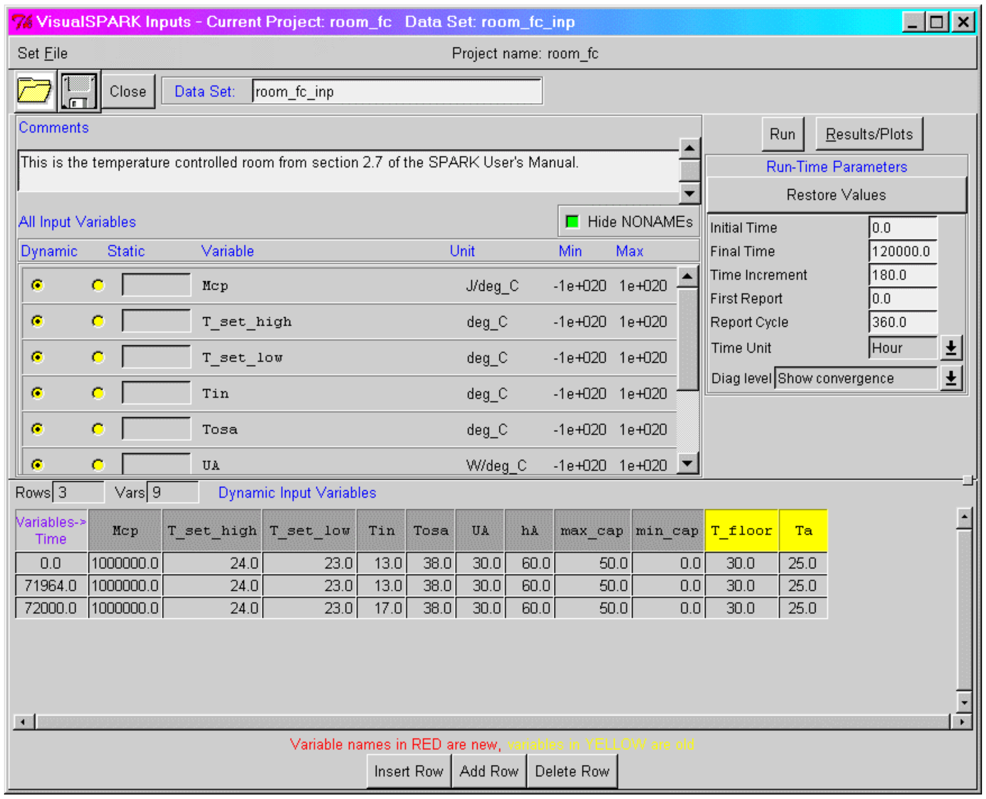
Q_floor is already selected for the X axis, so press the radio button under the “Y” for T_floor to select T_floor for the Y axis. Now press the “Make Graph” button near the bottom of the panel to create the graph:



Here we can see that they are more or less linearly related at the start but as Q_floor approaches 0, T_floor suddenly increases and then is flat while Q_floor increases until about 50, at which point they are again linearly related. The green-filled circle indicates the first data point and the blue-filled circle the last data point.

A.3 Modifying the Input Data

Let's play a little with the input data to see how it affects the model. Click on the "Edit Input Set" button in the main panel. If it is grayed out (inactive) then you must reselect the room_fc_inp dataset in the Projects area. Here is the input data editor:



VisualSPARK Inputs - Current Project: room_fc Data Set: room_fc_inp

Set File   Project name: room_fc

Close   Data Set: room_fc_inp

Comments

This is the temperature controlled room from section 2.7 of the SPARK User's Manual.

All Input Variables   Hide NONAMEs

Dynamic	Static		Variable	Unit	Min	Max
●	○		Mcp	J/deg_C	-1e+020	1e+020
●	○		T_set_high	deg_C	-1e+020	1e+020
●	○		T_set_low	deg_C	-1e+020	1e+020
●	○		Tin	deg_C	-1e+020	1e+020
●	○		Tosa	deg_C	-1e+020	1e+020
●	○		UA	W/deg_C	-1e+020	1e+020

Run   Results/Plots

Run-Time Parameters

Restore Values

Initial Time	0.0
Final Time	120000.0
Time Increment	180.0
First Report	0.0
Report Cycle	360.0
Time Unit	Hour
Diag level	Show convergence

Rows 3   Vars 9   Dynamic Input Variables

Variables-> Time	Mcp	T_set_high	T_set_low	Tin	Tosa	UA	hA	max_cap	min_cap	T_floor	Ta
0.0	1000000.0	24.0	23.0	13.0	38.0	30.0	60.0	50.0	0.0	30.0	25.0
71964.0	1000000.0	24.0	23.0	13.0	38.0	30.0	60.0	50.0	0.0	30.0	25.0
72000.0	1000000.0	24.0	23.0	17.0	38.0	30.0	60.0	50.0	0.0	30.0	25.0

Variable names in RED are new, variables in YELLOW are old

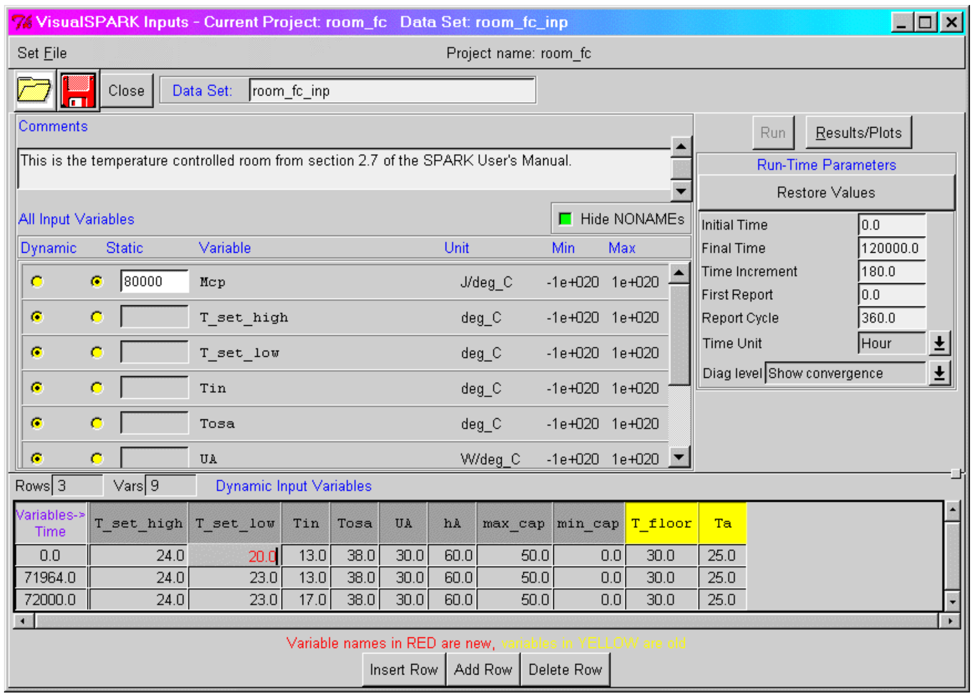
Insert Row   Add Row   Delete Row

In the upper section is a *Set File* menu to load and save a dataset file, and icons below that which do the equivalent. Below that is an area where arbitrary comments may be inserted, which are saved with the dataset file. Further down is the section that lists all the input variables for the problem. The radio buttons labeled "Dynamic" and "Static" tell the solver whether the value may change with time (dynamic) or is fixed (static). If the "Static" button is pressed for any variable then the box to the right becomes active, allowing you to enter a value for that variable, which it will have for the entire run.

On the right side are the Run-time parameters: initial time, final time, time increment, time of first reporting, time between reports, time units and diagnostic level. The diagnostic level is described in section 3.11.5 of the SPARK User's Manual

The bottom section contains a table of the values for all the input variables that are checked as "dynamic". The width of the columns may be manually changed to show longer variable names or larger values, as is the case here with the values for the variable "Mcp". To change the width of a column, click the right mouse button on the vertical line between the variable name cells and drag the mouse right or left to resize it. You should see a "+" cursor as you hold down the right mouse button.

Here's where we'll make some changes to see how it affects the results. Simply click on a cell and replace the value. Let's make the "Mcp" variable a static variable, and change the "T_set_low variable" at time 0.0 from 23.0 to 20.0:



VisualSPARK Inputs - Current Project: room_fc Data Set: room_fc_inp

Set File   Project name: room_fc

Close   Data Set: room_fc_inp

Comments

This is the temperature controlled room from section 2.7 of the SPARK User's Manual.

All Input Variables   Hide NONAMEs

Dynamic	Static		Variable	Unit	Min	Max
○	●	80000	Mcp	J/deg_C	-1e+020	1e+020
●	○		T_set_high	deg_C	-1e+020	1e+020
●	○		T_set_low	deg_C	-1e+020	1e+020
●	○		Tin	deg_C	-1e+020	1e+020
●	○		Tosa	deg_C	-1e+020	1e+020
●	○		UA	W/deg_C	-1e+020	1e+020

Run   Results/Plots

Run-Time Parameters

Restore Values

Initial Time	0.0
Final Time	120000.0
Time Increment	180.0
First Report	0.0
Report Cycle	360.0
Time Unit	Hour
Diag level	Show convergence

Rows 3   Vars 9   Dynamic Input Variables

Variables-> Time	T_set_high	T_set_low	Tin	Tosa	UA	hA	max_cap	min_cap	T_floor	Ta
0.0	24.0	20.0	13.0	38.0	30.0	60.0	50.0	0.0	30.0	25.0
71964.0	24.0	23.0	13.0	38.0	30.0	60.0	50.0	0.0	30.0	25.0
72000.0	24.0	23.0	17.0	38.0	30.0	60.0	50.0	0.0	30.0	25.0

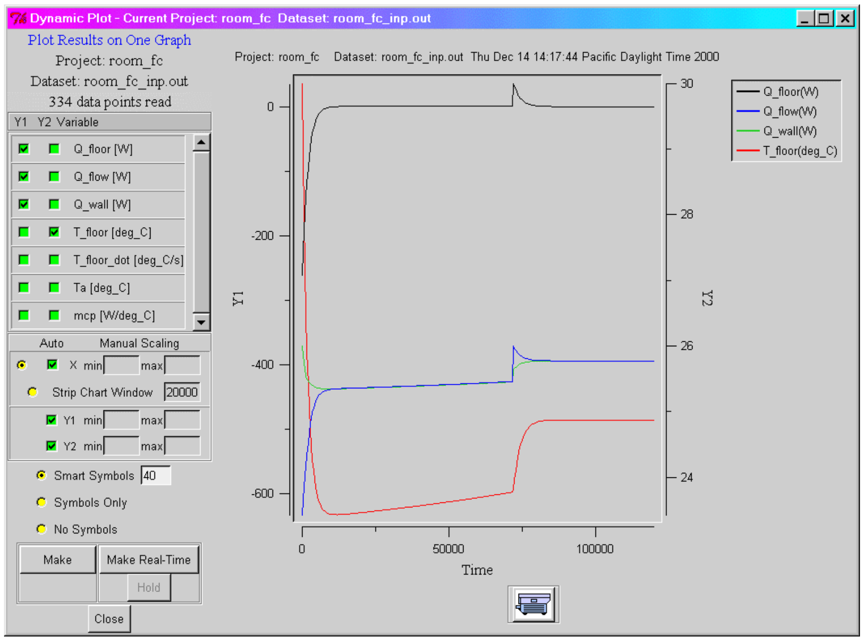
Variable names in RED are new, variables in YELLOW are old

Insert Row   Add Row   Delete Row

Here the “Mcp” variable has been changed from a dynamic (time-varying) variable to static by pressing the radio button under the “Static” label. Then a value was entered in the space to the right to give it a value that will be used throughout the run of the model. Notice that it has been removed from the “Dynamic Input Variables” table at the bottom.

To change a value of a dynamic variable (“T_set_low” in this case), simply click on the value in the cell and enter a new value. Notice that the 20.0 for “T_set_low” is red (it looks gray in printed documents), indicating that it has been changed. Also notice that the floppy disk icon at the top changed to red (also looks gray in printed documents) indicating that something has changed in the data, and the “Run” button has been disabled, i.e., clicking on “Run” doesn’t do anything. This is a safety to ensure that you save any changes before making a run.

Let’s look at the graph again:



Dynamic Plot - Current Project: room_fc Dataset: room_fc_inp.out

Plot Results on One Graph
Project: room_fc
Dataset: room_fc_inp.out
334 data points read

Y1	Y2	Variable
✓		Q_floor [W]
✓		Q_flow [W]
✓		Q_wall [W]
	✓	T_floor [deg_C]
		T_floor_dot [deg_C/s]
		Ta [deg_C]
		mcp [W/deg_C]

Auto / Manual Scaling
[✓ ●] X min max
○ Strip Chart Window 20000
✓ Y1 min max
✓ Y2 min max

● Smart Symbols 40
○ Symbols Only
○ No Symbols

Project: room_fc Dataset: room_fc_inp.out Thu Dec 14 14:17:44 Pacific Daylight Time 2000

You can see the difference compared to the graph on page 29.

Appendix B
SPARK Systems Programming

B.1 Overview

This appendix provides information on the SPARK build process, whereby the problem and class source files are converted to executable simulation programs. In addition to a description of the basic build process, a detailed description of the make file used to automate the build with the GNU make program is presented. All system and SPARK programs used in the build, and involved source files, are identified and described. Finally, the problem preference, file created during the build and used for problem execution, is discussed.

This information will be useful for advanced users or systems programmers who need to adapt the build process for different hardware platforms, or different operating systems. It can also be used to guide users who wish to use their own software tools, such as integrated development environments, to build SPARK problems.

B.2 Steps in SPARK Processing

Rather than solving a problem directly, SPARK builds a program that carries out the solution. This approach is taken in an effort to maximize solution efficiency. The overall process of building a solver and solving the problem is discussed in this section. Several system programs are used in this process, including your make program, compiler, and linker, as well as elements of the SPARK package.

SPARK processing for problem solution has five steps:

1. **Parse:** Parse the problem definition.
2. **Setup:** Develop a solution sequence.
3. **Compilation:** Compile the solution sequence.
4. **Linkage:** Link fixed and problem-specific binaries.
5. **Execution:** Numerical solution.

A diagram for this process is shown in Figure 22.

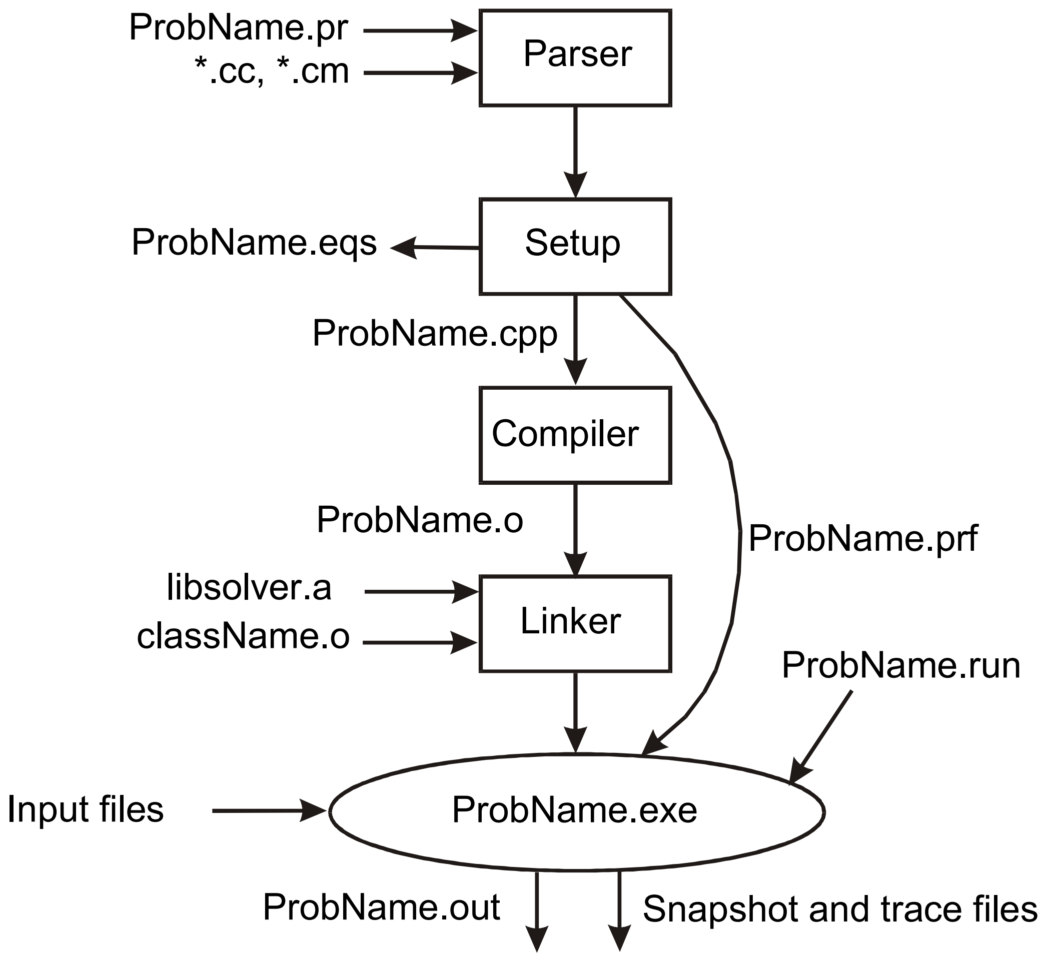


Figure 22 SPARK build process.

First, the *parser* program reads the problem definition file, *probName.pr*. The parser command line is:

```
    parser  [options]  <parseFileName>
```

where <parseFileName> gives the name of the file to be parsed, which can be either a problem, *probName.pr* or a class *className.cm*.

Parser options can include:

-p <class_path>: class_path is a list of directories separated by commas, e.g., c:\vspark\hvactk, c:\*vspark**myClasses*. These directories are searched in the given order to find classes needed by the problem. If omitted, the environment variable SPARK_CLASSPATH will be used in the same manner.

-e<error level>: Sets level at which parser will report errors. E.g., e1 will result in report of errors only. Higher numbers give increased verbosity: 1=errors, 2=warnings, 3=information, 4=more information. The default is e2.

-s<setup file verbosity>: Sets level of verbosity in the created setup file. Lowest is s1, with increasing verbosity up to s6. The default is s2.

-l < logFileName>: If -l < logFileName> is specified, the diagnostic messages are sent to logFileName. If -l is given without logFileName, the messages are sent to parseFileName.log.

-d <debug dump>: Extensive debugging output will be generated.

As the *probName.pr* file is parsed, various declarations of atomic and macro classes are found. The specified class paths are searched for these classes. If a macro class is found, it is parsed, and if the macro refers to other macros, they are parsed. This continues recursively to any depth. The result is to resolve all macros to atomic objects, so that at run time SPARK can operate entirely on atomic classes. This exposes individual equations and variables to the graph theoretic algorithms for organizing an efficient solution sequence.

The principal output of the parser is the setup file, *probName.stp*. This file is a different expression of the problem, with the principle difference being that the macros have been resolved.

The next step is carried out by the *setupcpp* program. The command line for *setupcpp* is:

```
setupcpp probName <enter>
```

Here, *probName* is the setup file produced by the parser (the *.stp* extension is omitted here).

The main function of *setupcpp* is to perform the graph theoretic analysis of the problem, producing a compact, efficient solution procedure expressed in *C++*. This is emitted as *probName.cpp*. If a typical *probName.cpp* file is examined (see *sum.cpp* in *vspark\doc\examples\2sum*, for example) it will be seen to consist entirely of definitions that set problem size, and data structure initializations.

The principal data structure in *probName.cpp* is the Component array, which defines the solution sequence for each strongly connected component in the problem. The solution sequence is a list of pointers to the object inverse functions identified by the graph theoretic algorithms in *setupcpp*. Thus at run time, instead of traversing the problem graph, the solver can simply step through these arrays (one for each component), invoking the referenced functions in the correct order. If the component is cyclic, the sequence is iterated in the normal Newton-Raphson manner.

In addition to the *probName.cpp* file, *setupcpp* produces two other files. One is essential to final problem execution, namely *probName.prf*. This file serves to transmit needed information from the setup process to the solution process. It is discussed in a subsequent section. The other produced file is *probName.eqs*. This file, discussed in Section 2 of the User's Manual, is a human-readable expression of the solution sequence. It is not used by SPARK, but is sometimes helpful in debugging when numerical difficulties are encountered.

The next two steps in the process are compilation and linkage. The *probName.cpp* file must be compiled, as well as needed inverse functions from the objects in the problem. Then these parts, which change when the problem structure is changed, are linked with the fixed SPARK procedures. In particular, they are linked with the SPARK main[13] program. The result of the linkage is the "solver," *probName.exe*, which solves the problem.

Finally, the solver is executed to solve the problem with the command line:

```
probName probName.prf probName.run
```

This executable will read the input files specified in the run control file probName.run. Otherwise, values for *input* variables will be taken from *init* or *default* values specified in the problem The User's Manual should be consulted for more details on the SPARK initialization mechanism.

[13] See *int main(unsigned argc, char **argv)* in *main.cpp* if you have the SPARK source code.

B.3 SPARK Source Files and Build Targets

Normally, users need not be concerned with SPARK source files because the build process is automated as described below (See Automating the Build Process). However, if you wish to build a solver manually, or with the aid of an integrated development environment, you will need to know about the fixed, binary files and the source files used by SPARK.

The main target of a build is the executable file, called the *solver*, customarily renamed *probName.exe*. The binaries for the fixed code files that must be linked into every SPARK solver are archived in *libsolver.a*[14] that can be found in the *vspark\lib* directory.

The files shown in Table 3 are the additional files needed to make the executable for a particular problem, *example.pr*. There will always be a *probName.cpp* file, plus a *className.cc* file for every atomic class declared in *probName.pr* or declared in any macro class used anywhere in the problem. The *probName.cpp* file is produced by the *setupcpp* program. The names of the needed atomic classes are determined by the *parser* and written at the end of the *probName.stp* file.

Compilation also requires certain header files. These are stored in the *vspark\inc* directory.

Table 3 Problem Dependent Code for example.pr

example.cpp	r1.cc	r2.c	r3.cc	r4.cc

B.4 Automating the Build Process

The SPARK build process described above is rather complex. As has been noted, the SPARK objects used in the problem are not known until the *probName.pr* file has been parsed. Moreover, the particular inverse functions from these objects that will actually be used in the numerical solution process are not known until *setupcpp* has completed. Thus the files to be compiled and linked are not knowable beforehand. Finally, when the user makes a change to the *probName.pr* file or any used class files, the build process should remake only affected files.

To simplify matters for users the process of building a SPARK solver is automated using facilities of the GNU *make* program, *gmake,* and a *makefile* called *makefile.prj,* found in *vspark\lib*.

B.4.1 Using gmake to Run a SPARK Problem

From the usage standpoint, *gmake* is straightforward. It will build an executable program, i.e., a solver called *probName*, from the specified *probName.pr* source file and then execute it. To do so, it uses the local symbolic link *makefile* which points to *vspark\lib\makefile.prj.* Since this symbolic link is in the current working directory, along with *probName.pr* and needed *probName.inp* files, you can build a solver for a particular problem by entering

```
    d:\vspark\probName> gmake run <enter>
```

This is the most common usage of *gmake* in the SPARK context. However, there are other command line targets to meet special needs. These are shown in Table 4.

Table 4 gmake Options

Command line Targets	**Purpose**
run	To make a run using PROJ.inp as input and PROJ.run as run-control file. If PROJ.run is missing, a default one is created.
(none)	After the first time, rebuilds the 'makefile.inc' and the solver executable as needed using the previous SPARK_CLASSPATH.
PROJ=xxxx SPARK_CLASSPATH=..\class	To create the file 'makefile.inc' and the solver

[14] The *VisualSPARK* Windows distribution currently provides binaries for the mingw C++ system only.

	executable that has the same name as PROJ.
Solver	To make the solver executable, i.e., PROJ
stp	To make PROJ.stp file.
makefile.inc	To create *makefile.inc* file.
pkg	To create an export package in ..\PROJ_pkg directory.
prf	To make PROJ.prf file.
clean	To delete all intermediate files.
cleanALL	Same as clean, but also deletes all run subdirectories.

B.4.2 SPARK Makefiles

The overall approach to doing automatic builds for SPARK problems has been described in previous sections of this Appendix. In this Section we examine a simplified version of the *makefile.prj*[15] file so that the process may be understood more fully. The *makefile.prj* file discussed here is shown below.

B.4.2.1 The makefile.prj *File*

```
#
# Define project base name
#
PROJ := $(shell basename `pwd`)
M_PROJ := $(PROJ)
#
# Set class path
#
ifndef  SPARK_CLASSPATH
  SPARK_CLASSPATH :=
.,../class,$(SPARK_DIR)/globalclass,$(SPARK_DIR)/hvactk/class
endif
#
# Include variable part of makefile. This file defines class
# dependencies. It is not available in first make invocation,
#  but it gets created by the make process.
#
include  makefile.inc
# Define paths to SPARK binaries
LIBFILES := $(SPARK_DIR)/lib/libsolver*.a
PARSER :=   $(SPARK_DIR)/bin/parser
PARSER_FLAGS := -l -p "$(SPARK_CLASSPATH)"
SETUP :=    $(SPARK_DIR)/bin/setupcpp
MKMAKINC := $(SPARK_DIR)/bin/mkmakinc
#
# Define system binaries and flags
#
CPP :=      g++
CPPFLAGS := -g -w -I$(SPARK_DIR)/inc
LDPP :=     g++
LDPPFLAGS := -g
#
# Dependency and make rules
```

[15] The actual *makefile.prj* file may be seen in the *vspark\lib* directory. The version discussed here has been simplified for clarity.

```
#
DEPsO := $(M_DEPsCC:%.cc=%.o)
%.o:  %.cc
      rm -f $@
      $(CPP) -c -o $@  $(CPPFLAGS)  $<
%.o:  %.cpp
      rm -f $@
      $(CPP) -c -o $@  $(CPPFLAGS)  $<
#
# Reinvoke gmake to make solver
#
solver:     makefile.inc
      $(MAKE)  $(M_PROJ)
#
# Reinvoke gmake to run the solver
#
run:  makefile.inc
      $(MAKE)  $(M_PROJ).out
#
# Make solver
#
$(M_PROJ):  $(M_PROJ)_.o  $(DEPsO)
      rm -f  $@
      $(LDPP) -o $@  $(LDPPFLAGS)  $^  -lm
#
# Compile project C++ file
#
$(M_PROJ)_.o:     $(M_PROJ).cpp
      rm -f $@
      $(CPP) -c -o $@  $(CPPFLAGS)  $<
#
# Run setup. Note multiple targets.
#
prf  $(M_PROJ).prf  $(M_PROJ).cpp:  $(M_PROJ).stp)
      rm -f $@  setup.log
      $(SETUP)  $(M_PROJ)   > setup.log  2>&1
#
# Run parser.
#
stp  $(M_PROJ).stp:  $(M_PROJ).pr  $(M_DEPsCM)  $(M_DEPsCC)
      rm -f   $(M_PROJ).stp $(M_PROJ).log
      $(PARSER) $(PARSER_FLAGS)  $<
#
# Run solver.
#
$(M_PROJ).out:    $(M_PROJ) $(M_PROJ).prf $(M_PROJ).run
      rm -f   $@  run.log
      ./$(M_PROJ)  $(M_PROJ).prf  $(M_PROJ).run  > run.log  2>&1
#
# Determine needed classes and create the include file.
#
makefile.inc:     $(M_PROJ).pr  $(M_DEPs_CM)
      rm -f   $(M_PROJ).stp $(M_PROJ).log $(M_PROJ).tmp
      -$(PARSER) $(PARSER_FLAGS)  $<
      if [ -r $(M_PROJ).stp ] ; then       \
          Stp=$(M_PROJ).stp         ;\
      elif [ -r $(M_PROJ).tmp ] ; then \
          Stp=$(M_PROJ).tmp         ;\
      else                     \
          exit 1               ;\
      fi                       ;\
	rm -f makefile.inc           ;\
	$(MKMAKINC)  $$Stp  >  makefile.inc ;\
	echo "SPARK_CLASSPATH=" $(SPARK_CLASSPATH)  >> makefile.inc
```

B.4.2.2 Explanation

The first important observation in the above make file is the "include" statement, a special feature of GNU make analogous to the C include directive. Upon the first *gmake* execution for a particular project, there will be no *makefile.inc* in the project directory. This generates a warning report, but otherwise is of no consequence. However, in a later, recursive invocation of *gmake* using the same *makefile.prj*, the include file will be created. Consequently, in still-later recursive invocations of *gmake*, again with *makefile.prj*, the *makefile.inc* will be present, thus altering *makefile.prj*. This is important, as it allows the SPARK build process to properly deal with the initially unknown list of classes to be compiled and linked.

With this understanding, let us consider a typical SPARK project build. The assumed target is the pseudo target *run*. Since pseudo targets are always considered out-of-date or absent by g*make*, this target will always need processing. However, the dependency list is the include file, *makefile.inc*. Therefore this is a mechanism for forcing g*make* to build *makefile.inc*. Afterwards, it will continue with the command for the *run* target, which is seen to be a recursive invocation of *gmake* with the project out file, i.e., numerical results, as the target. We shall come back to examine this after we deal with the *makefile.inc* dependency.

If we look at the *makefile.inc* target, we see dependencies of $(M_PROJ).pr and $(M_DEPs_CM). The former is the problem file, assumed to be in the working directory. The latter is a string macro that is defined in the *makefile.inc*. Now, since this file is not initially present, M_DEPs_CM will be undefined so *gmake* will return a null string when it is evaluated; this is as if this dependency was not listed. At any rate, since makefile.inc is not present, it is by definition "outdated" and the make commands will be executed. There are basically two steps here: first the parser is executed, creating the *PROJ.stp* file, then the special SPARK program called *mkmakeinc* is executed. This program interrogates the *PROJ.stp* file to find the list of atomic (.cc) and macro (.cm) classes used in the problem, writing the list out to the *makefile.inc* file. Afterwards, the UNIX echo command is used to append the class path string at the end, thus creating a makefile.inc as exemplified below:

```
M_PROJ=   projectName
M_DEPsCC= ../../globalclass/atomic1.cc  ../class/atomic2.cc etc.
M_DEPsCM= ../../globalclass/macro1.cm  \ ../class/macro2.cc etc.
SPARK_CLASSPATH= ".,$(SPARK_DIR)/globalclass"
```

We can now return to the command line for the *run* target, where we see a recursive invocation of *gmake* with the target of $(M_PROJ).out. This target has dependencies of the solver executable $(M_PROJ) and the preference file $(M_PROJ).prf. Assuming these to be out of date, *gmake* will build them before running the $(M_PROJ).out target command.

The $(M_PROJ) target has dependencies of $(M_PROJ)_.o and $(DEPsO). The former is the compiled project C++ file. Following the *makefile* to this target shows that it is built by running the C++ compiler, but only after the *setupcpp* program is executed to create $(M_PROJ).cpp. Presumably, this build thread stops here, since the dependency list for $(M_PROJ).cpp is only $(M_PROJ).stp, which should be up to date since the parser was just run. Therefore the *setupcpp* program is executed, then the compiler to produce $(M_PROJ)_.o.[16]

We can now turn our attention to the second dependency of the $(M_PROJ) target, $(DEPsO). This string macro is defined to be the list of compiled atomic objects. This definition is accomplished by use of macro string substitution based on M_DEPsCC as defined in *makefile.inc*. The consequence of this is an implied list of *.o* targets, which are then built using the *.o:.cc* inference rule provided in *makefile.prj*, i.e., execution of the C++ compiler. When this is complete, finally the rule for building $(M_PROJ) can be executed. This is a call to *g++*, acting in this case as a linkage loader.

Once $(M_PROJ) has been created (or updated, as the case may be), the $(M_PROJ).out target command can be executed. This is seen to be simply execution of $(M_PROJ), with arguments $(M_PROJ).prf and

[16] The underscore is to prevent possible collisions with atomic classes of the same name as the project.

$(M_PROJ).run. The latter provided run-control information, as explained elsewhere in this Guide and in the User's Manual. The actual *makefile.prj* has $(M_PROJ).run as a target with commands for building a default version of this file. We omitted it here for simplicity.

This concludes the original, command line invocation of *gmake* with *makefile.prj* as the makefile. Details were omitted, but perhaps the general ideas will be sufficient for users who having special needs that require modifications of the build process.

B.5 Problem Preference Files

Each SPARK problem has an associated preference file that sets important information needed by the solver. This file describes the settings for the numerical solution of each component of the problem . In addition, this preference file includes a list of the *C++* source files that are specific to the problem. As explained earlier, the problem-preference file, *probName.prf,* is generated by the SPARK *setupcpp* program at the same time that it generates *probName.cpp*. The following preference sfile is for the *example.pr* problem:

```
(
 ComponentSettings (
   0 (
         ComponentSolvingMethod ( 0 ())
         TrueJacobianEvalStep ( 1 ())
         Epsilon (1.E-6 ())
         RelaxationCoefficient ( 1.0 ())
         ScalingMethod ( 0 ())
         MaxIterations (50 ())
         Tolerance (1.E-6 ())
         AbsTolerance (1.E-6 ())
         MaxTolerance (1.E-3 ())
         MatrixSolvingMethod ( 0 ())
         PivotingMethod ( 1 ())
         RefinementMethod ( 0 ())
   )
)
Sources (
   ./example.cpp ()
   ../class/r1.cc ()
   ../class/r2.cc ()
   ../class/r3.cc ()
   ../class/r4.cc ()
)
)
```

Since *probName.prf* is a text file, any text editor can be used to edit it. Alternatively, you can use tools provided with SPARK. One of these tools is a command line program called *repref*. In general, execution of *repref* is as follows:

```
repref <file.prf> <pref 0> <pref n-1> <action> <key>
```

This modifies the branch <pref 0> ... <pref n-1> according to <action>:

= <key> -- replaces value at the branch by <key>

- <key> -- removes value <key> value at the branch

+ <key> -- add value <key> at the branch

As an example, to change Epsilon for Component 0 in *example.prf*:

```
repref example.prf ComponentSettings 0 Epsilon = 1.e-8 <enter>
```

Repref is handy for writing script files for preference file modifications.

Glossary

atomic class

A model comprising a single equation with used variables linked to its ports. Acts as a template for instantiation of atomic objects.

class

A general description of an equation (atomic class) or group of equations (macro class). A class acts as a template for instantiation of objects.

command file

A file containing MSDOS commands. Also called a "batch" file .

cyclic

In graph theory, the property of having closed paths, or circuits.

environment variable

A symbol whose value is assigned in your computing environment, as opposed to within the SPARK program system. See documentation for Microsoft Windows for more information and to learn how environment variables are set. See also *sparkenv*.

GNU

Gnu is Not UNIX. A system of free software programs developed through the Free Software Foundation.

HVAC

Heating, ventilation, and air-conditioning.

input set

The complete set of information needed to define execution of a SPARK problem. Includes input data files and run control information.

instantiate

To create an object based on a class definition. The *declare* keyword in SPARK performs instantiation.

inverse

A formula obtained by symbolic solution of an equation for a particular variable in the equation. An explicit inverse has the wanted variable on the left side only, while an implicit inverse has that variable in the formula as well.

macro class

A group of SPARK atomic or other macro classes linked together through their respective ports to form a subsystem model. A macro class can be used wherever an atomic class can be used.

make

A utility program that creates a program from its composite parts, in response to commands embedded in a makefile. GNU make is used for the UNIX implementation of SPARK.

makefile

An input file for a *make* program. Contains various targets, their dependencies, and commands for building them.

panel

A discernible region within a window on your computer screen.

parser

The program that interprets the SPARK input files as the first step toward solution. Builds the setup file.

PDF

A portable file format from Adobe Systems that retains page layout and graphics. You need a special program, called Acrobat Reader, to view a file in PDF format. This program is freely available on the Internet.

preference file

A file that contains various information needed for a program to run. In a sense, a generalization of command line options and environment variables.

run-control

Data controlling the solution phase for a SPARK problem, e.g., start time, finish time, and time step.

setupcpp

A program used in the process of building a SPARK problem. Processes the setup file.

solver

The executable program that SPARK builds to solve a particular problem. Called probName.exe (Windows) or probName (UNIX). The underlying programs used by SPARK in building the executable program are sometimes referred to as the "solver" as well.

sparkenv

A command file for setting up your environment for running SPARK at the command line.

spawn

To create a computational process in a computer.

strongly connected component

In graph theory, a maximal set of vertices and edges that allow any vertex to be reached from any other vertex. In SPARK, corresponds to a separately solvable subproblem.

symbolic

Operations on mathematical expressions in terms of contained symbols, as opposed to numerical evaluation. The goal is often to solve for a particular variable in terms of all others in the expression, i.e., obtain an inverse.

target

A file or other object that can be created with one of the command sequences in a makefile.

tool bar

A row or column of icons, usually at the top of a window, that can be clicked to perform commonly need tasks. The icons usually are pictorial, suggesting what the tool does. For example, the Print icon on many *VisualSPARK* windows looks like a laser printer.

VisualSPARK

An implementation of SPARK, together with a graphical user interface for the Microsoft Windows and the UNIX operating systems. The *VisualSPARK* graphical user interface is copyrighted by the Regents of the University of California.

WinSPARK

An implementation of SPARK, together with a graphical user interface for the Microsoft Windows operating system. The *WinSPARK* graphical user interface is copyrighted by the Ayres Sowell Associates, Inc.

Index